ScriptEase [tm]

ISDK for C/C++

v 4.20

Nombas, Inc.

SE ISDK/C 4.20 Manual, 3rd edition

Editor: Bill Ross

Nombas, Inc.
64 Salem Street
Medford, MA 02155
USA

http://www.nombas.com/

Table of Contents

Introduction
To the ScriptEase:Integration SDK/C

The JavaScript interpreter included in this SDK allows you to integrate the JavaScript language with your application. Your end users will then be able to write macro scripts in JavaScript to customize your application to their needs. Compact and stable, the ScriptEase:ISDK/C is easily integrated into your application with straightforward and intuitive function wrappers and initialization calls.

The basic syntax and commands of JavaScript are determined by the ECMAScript standards committee. Nombas' JavaScript has several enhancements that are not required by the ECMAScript standard, but do not interfere with the running of regular JavaScript. These enhancements include the preprocessor directives #include and #define, the switch and case statements, and dynamic object handling. Many of these commands are already familiar to programmers and are expected to be included in future versions of the JavaScript standard.

The SDK gives you the freedom to control exactly what you want accessible by the macros. You control how your customers may invoke scripts (e.g., with pull down menus, commands on a command line, hot keys, function keys, etc.) and what the macro scripts can do (manipulate data, position windows, interact with other applications, retrieve input, handle output, e.g.). Virtually anything that can be done with a compiled language can be done with ScriptEase. The possibilities are limited only by your imagination.

A number of sample applications come with ScriptEase:Integration SDK. These provide examples demonstrating how to develop an application with the Software Developer's Kit. Each sample demonstrates how to integrate the Integration SDK into your application. Often, within a sample application and between samples, multiple solutions demonstrate the SDK's versatility. This information is also in this manual, but we know that sometimes the best way to understand something is to look at working code. We also provide wrapper functions for all of the standard ECMAScript objects, browser objects and the standard C library (plus a few additional ones that deal with files, directories and script execution) as source codes.

SE:ISDK/C Versions and Contents

While the ScriptEase:ISDK/C is available in versions for DOS, OS/2, 16 bit and 32 bit Windows, Macintosh and various UNIX flavors, other platforms are available and custom environments may be ordered. The API and programming descriptions in this manual are valid regardless of which platform you are developing for.
Included with each order are the ScriptEase Standard Library source files and numerous sample applications. Feel free to use the included library and sample code as examples for study or include them in your application.

If you have any questions or comments, please contact:
Nombas - ScriptEase:Integration Software Developer's Kit Development Team
E-mail: toolkit_dev@nombas.com
Phone: 781-391-6595
FAX: 781-391-3842

What's New

in ScriptEase ISDK/C **4.20**

SE 4.20 continues Nombas' commitment to updating and improving our ScriptEase Integrated System Development Kit.

Quick List

Garbage Collection

- new API call, jseGarbageCollect()
- no cyclic flags
- improved performance

Internationalization

New function - jseGetResourceFunc

MBCS Support

For **multibyte character sets** (MBCS) other than Unicode.

Exception Handling

- try/catch/throw exception handling
- New Error objects, allowing for the application to be called back at the point an error was generated.

New Operators

- instanceof operator
- Strict equality (=== and !==)

Reg Exp

Regular Expressions now a standard ECMAScript object.

New ECMAScript functions

• Array.concat • Array.pop • Array.push • Array.slice • Array.splice • Array.unshift • Date.toDateString • Date.toLocaleDateString • Date.toLocaleString • Date.toLocaleTimeString • Date.toTimeString • Function.apply • Function.call • Object.hasOwnProperty • Object.isPrototypeOf	• Object.propertyIsEnumerable • Object.toLocaleString • String.concat • String.localeCompare • String.slice • String.toLocaleLowerCase • String.toLocaleUpperCase • Error • ConversionError • EvalError • RangeError • ReferenceError • RegExpError • SyntaxError • TypeError

New #link Objects

- **Database rewritten**

  ScriptEase Database Connectivity (SEDBC) is a rewrite of the database access object. SEDBC is object oriented and much easier to use than the previous ODBC database object.

  Also new is the SimpleDataset object, which provides basic database access and modification without requiring any knowledge of SQL or database concepts.

- **COM object** for Windows COM model

  Makes the Microsoft COM model look like a JavaScript object.

  This code is a good demonstration of how dynamic objects are used to map JavaScript to other object models.

New Flag

jseOptToBooleanObjectEval has been added to the jseLinkOptions.

According to ECMAScript, ToBoolean() on an object always results in "true". This new run-time flag allows that behavior to be altered.

New Initializer syntax

- Object initializers {a:4,b:2}
- Array initializers [1,2,3]

More on:

Multibyte Character Sets

An important addition to SE 4.20 is the support for **multibyte character sets** (MBCS) other than Unicode. Several customers have needed this support, and SE420 is being released primarily to support arbitrary multibyte characters sets. We have also made our internal strings easier to replace (presumably with a different language,) in order to better support internationalization.

Support For Forthcoming ECMAScript Spec.

There are other improvements as well. The ECMA committee is nearing completion of the next ECMAScript specification, and we have added support for a number of constructs that will appear in it. While several have been part of ScriptEase all along (such as the switch statement and do/while loop statements), we have added support for several other constructs. The try/catch/finally error-handling statement is now included, and errors are encapsulated in ECMA-compatible error objects.

We have updated and improved our array literal format to match the new ECMA specification, and added support for the object literal. Many new object methods have been added to the standard ECMA objects (such as String, Number, and Array). We have changed our regular expression external library to be part of the standard core distribution, now that it is included in the ECMAScript specification. We will continue to support the ECMAScript specification as it evolves and is finalized.

More Improvements

Internally, we have dropped our reference counting scheme in favor of a mark-and-sweep garbage collector. Memory is allocated in larger chunks now, reducing the overhead and thus overall memory usage. Several new compile-time macros can now be defined to get more exacting control over how memory is used, which is important for systems with little memory. In addition, the internal engine is no longer burdened with needing to track reference counts, which allows pure computational scripts to perform significantly better. None of these changes required any alteration of the ScriptEase ISDK API, so all programs will continue to work without modification.

To go along with these changes, improvements have been made to the ScriptEase API

itself. All of these changes are backwards-compatible, so you will not have to modify your application, just recompile and relink.

A new call has been added to force or disable garbage collection. jseInterpret() and jseCallFunction() now allow errors generated within them to be trapped instead of always being printed out. jseCallFunction() also makes it easier now to call a constructor function to generate a new object.

With all these improvements, ScriptEase version 4.20 is the best ScriptEase ever. Enjoy!

Integrating the ISDK/C

This chapter describes the methods for integrating the ScriptEase:ISDK/C into your application. Integration is comprised of three elements: jseContext, function wrappers, and jseVariables. All functions mentioned in this chapter are fully described beginning in the API chapter.

Unpacking, Installing ScriptEase:ISDK/C

The package you received consists of 3 basic parts: the various ScriptEase:ISDK/C Interpreter Engines, the ScriptEase Standard Function Library source code, and miscellaneous source files required to support the interpreter engine.

ScriptEase:ISDK/C Interpreter Engines are the static and dynamic linked libraries in the seisdk\libs directory. The ScriptEase:ISDK/C Interpreter Engine contains the complete interpreter (i.e. parser, interpreter, operator and flow-control commands), but does not include any function libraries.

You will find one or more interpreter engines in your installation tree. All of the engines interpret scripts identically; use the version that corresponds to your compiler and operating system.

Run the install or setup program to install ScriptEase:ISDK/C to your hard drive. This unpacks and copies all the source and sample code to the drive and directory you specify. Refer to the file README.TXT on the first installation disk for any last minute information and for more details on the installation procedure.

The install program will build a directory tree resembling the following:

- se420
  - incjse
  - srclib
    - ecma
    - common
    - clib
    - win
  - srcmisc
  - srcapp
  - seisdk
    - samples
      - simple0
      - simple1
      - simple2
      - simpedit
    - libs
  - srccore (if you have licensed source to the engine)

Integration overview

Integrating the SDK with your application is a simple and straightforward procedure. You must include the source files in your project or makefile, and make several directories available to your compiler.

You must also create a file called JSEOPT.H to include in your application's source file. This is a header file that defines certain flags the interpreter must use for proper operation and system identification. These definitions may be done on the compiler's command line if it is more convenient to do so, but the JSEOPT.H file must still exist, since the interpreter relies on it to operate.

To run the interpreter from within your application, it must first be initialized with jseInitializeEngine(). Then a jseContext for the scripting session must be created by calling jseInitializeExternalLink().

The jseContext defines the scripting session's global variables, functions, security, and operating parameters (such as error handling). Although usually one jseContext is sufficient, you may use more than one jseContext to create simultaneous and independent scripting sessions, with unique sets of global variables and available functions. When your application is finished scripting, these jsecontexts must be de-initialized by calling jseTerminateExternalLink() and jseTerminateEngine().

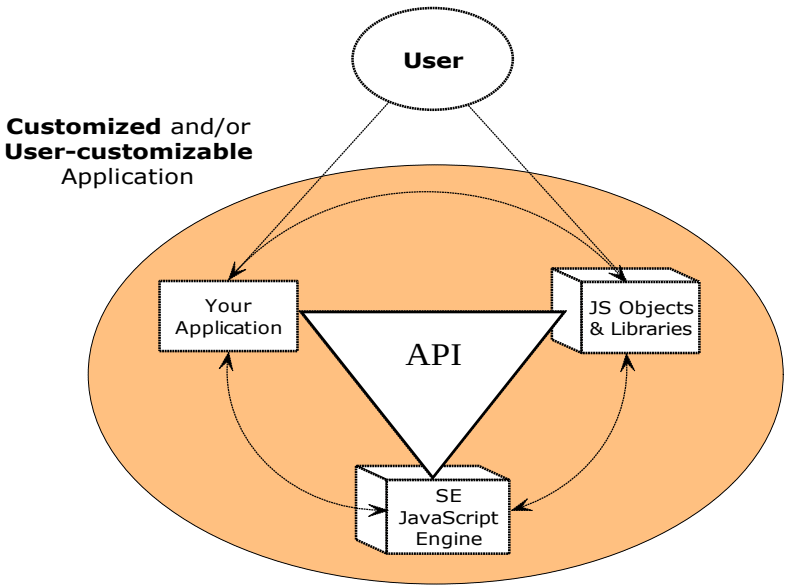
Finally, for each function you make available you must write a wrapper function that converts variable types from ScriptEase JavaScript to C and back. These functions must

then be assigned to the jseContext with calls to jseAddLibrary(). Nombas provides many pre-written wrapper functions which you may include in your projects.

Executing a script from within your application is done with a single call to jseInterpret().

The ScriptEase ISDK

Environment



You can use the ISDK to quickly and easily customize your applications.

Your application uses the SE JavaScript engine, and the libraries, to offer macros and customizing to your Users.

The ISDK's API (Application Programming Interface) gives you a variety of ways to interact with the JavaScript Engine.

Required source code and headers

The compiled ScriptEase libraries contain most of the core ScriptEase interpreter. You will need to set up your project or makefile to link with one of these libraries. Given the compact nature of the ScriptEase engine, we encourage customers to use the static libraries whenever possible as it makes your application easier to maintain.

There are several C/C++ source files that you must include with your project or makefile depending on the options selected in JSEOPT.H. These source files will be found in srcmisc for miscellaneous functions, srcapp for default callbacks an application must provide and the srclib tree for pre-written language objects and libraries.

JSEOPT.H

JSEOPT.H is a header file that defines the application's operating system and how it links with the interpreter. If you plan to use any of the Nombas supplied wrapper functions you must define the appropriate file in JSEOPT.H (A complete list of Nombas supplied functions can be found in the API chapter, and a complete list of Types and Macros can be found in that chapter, next). A JSEOPT.H file must be compiled with every application that uses the Integration SDK/C.

Your JSEOPT.H file defines the operating system with a __JSE_XXX__ define, where XXX refers to the target operating system. These are some of the common systems (more can be found in JSELIB.H). Note that each begins and ends with two underscore characters:

DOS	__JSE_DOS16__
DOS32	__JSE_DOS32__
OS/2	__JSE_OS2TEXT__ or __JSE_OS2PM__
16-bit Windows	__JSE_WIN16__
32-bit Windows	__JSE_WIN32__
UNIX	__JSE_UNIX__
MacOS (PPC & 68K)	__JSE_MACPPC__ or __JSE_MAC68K__

The JSEOPT.H file also specifies the method used to link with the ScriptEase interpreter. This will be a static library or a shared library, depending on the platform your application runs on: for OS/2, Windows and protected mode DOS it is a DLL or static library; for UNIX it is an ELF shared object; on the Macintosh Operating Systems it is a code fragment. Refer to the table below for the library/platform combinations available and the associated directives.

- **DOS:** Static Library
- **DOS32:** Static Library
- **OS/2:** Static Library or load-time DLL
- **Windows:** Static Library, load-time DLL or runtime DLL
- **UNIX:** Static Library
- **MacOS (PPC & 68K):** Static Library

If you are linking with the static library, you must define __JSE_LIB__.

If you are linking with the load-time DLL, you must define __JSE_DLLLOAD__.

If you are linking with the run-time DLL, you must define __JSE_DLLRUN__.

These directives can be omitted from the JSEOPT.H and specified on your compiler's command line if it is more convenient to do so, but an empty JSEOPT.H file must be included with your application's code.

In this file you may define any number of preset values to include functions from Nombas' libraries. After defining any options and functions to include, you must then include the file "seall.h". This will automatically include all the header files that you need and in the correct order.

Example of a JSEOPT.H file

Here is an example of a JSEOPT.H file for an application that allows its users to use the printf() and puts() functions of the standard C library.

```
#ifndef _JSEOPT_H
#define _JSEOPT_H
#define __JSE_WIN32__ // Build for Win32
#define __JSE_LIB__ // Link with static library
#define JSE_CLIB_ALL // Include all the Clib functions
#define JSE_CLIB_PUTS 0 // Don't include Clib.puts
#define JSE_LANG_ALL // Include all of the Language library
#include "seall.h"  // This will include everything for you
#endif
```

There are a large number of flags that can be modified to specify exactly what commands will be understood by the interpreter. These flags and their default values are listed in the chapter on Preprocessor Options.

Initializing the ISDK/C with your application

Before any scripting takes place, your application must initialize the ScriptEase interpreter by calling jseInitializeEngine(). This call starts off the entire scripting session. jseInitializeEngine() returns the version number of the ScriptEase:ISDK/C interpreter. Compare this with JSE_ENGINE_VERSION_ID to ensure that the proper library or DLL is installed.

jseContext

For each instance of scripting, you must create a jseContext to define the scripting session's operating environment and parameters. The jseContext provides a handle for all calls to and from the SDK/C. It sets up and begins the scripting session, defines its global data, routines for error handling, and security. You must specify a context whenever you get or set the value of a ScriptEase variable or request the state of the ScriptEase engine.

Contexts are dynamic; the ScriptEase processor will modify them while the script executes. For this reason, you must use the context supplied as a function parameter in your wrapper routines for all ScriptEase:ISDK/C calls, and not the one you initially create, unless instructed to do otherwise.

You may use several jseContexts in the same application. This is useful if you need scripts in different areas of your application and you don't want them to have access to the same functions and variables. Whenever you add ScriptEase libraries or variables, you add them to a specific context.

Creating a jseContext

Create a jseContext by calling jseInitializeExternalLink(). A jseContext will be returned. It is now ready for the functions used in your scripting session to be incorporated via jseAddLibrary(). Save this context; you will need it to terminate the link to the interpreter at the end of the scripting session.

jseIntializeExternalLink() takes four parameters: LinkData, a pointer to any data you wish to make available to scripts using the jseContext you are creating; LinkParms, a structure whose members specify various aspects of the scripting session; Global Name, the name for the top-level global variable in that context; and Userkey, your license ID.

LinkData is a "cookie" which provides you with a way to pass information to the routines you supply to a ScriptEase context.

jseExternalLinkParameters is a pointer to a structure defining the context parameters and options of the scripting session. It is fully described in the next section.

The jseExternalLinkParameters structure

The jseExternalLinkParameters structure defines the options that will be in effect during the scripting session. Its fields include pointers to functions that specify the data access, error messaging, breakpoint control, and variable scope.

Required

jsePrintErrorFunc is called by the interpreter when an script encounters an error condition. It will be called before the function cleans up the error. **errorString** is the

message associated with the error. This may be set to *NULL* for no error messaging (i.e., no function will be called before clean up).

Nombas provides a default version of this function, called ToolkitAppFileSearch, which is found in the file srcapp\fsearch.c. Be sure to include the header file fsearch.h for a prototype of this function (This header is automatically included with seall.h).

For Optional File I/O

jseFileFindFunc and jseGetSourceFunc, described below, together form the file access system for your scripts. The most common implementation would just locate source files and read them line by line; however, you may configure them to find and read any type of data from any local or remote source.

jseFileFindFunc translates whatever was supplied as a data source into a filename (or other address) the interpreter can use. A simple implementation would just make sure that the file specification received is complete and valid; it could also prevent access to certain files or directories or translate arbitrary strings into valid filenames. jseFileFindFunc is called before the interpreter begins reading from the source file passed to jseInterpret(), before reading a file specified by a #include directive, and before linking with a file defined by the #link directive.

> **FileSpec** is a *NULL*-terminated string representing a partial representation of the file. This will be whatever was passed to jseInterpret() or what was in the #include or #link spec.
>
> **FilePathResults**: A buffer where your application can store the results of the filefind, if a file was found. If this function returns *True* then this will be the value passed to the jseGetSource() **jseNewOpen** call.
>
> **FilePathLen**: the size of the **FilePathResults** buffer.
>
> **FindLink**: Set this to *True* of the file sought is part of a #link statement; otherwise it should be set to *False*.

jseGetSourceFunc tells the interpreter how to open the files for internal use. The file information is held in the jseToolkitAppSource structure, while flags specifies the action that is to be taken on the file.

```
typedef jsebool (JSE_CFUNC FAR_CALL *jseGetSourceFunc)
                (jseContext, jsecontext, struct,
                 jseToolkitAppSource, jseToolkitAppFlags, flags);
```

The **jseAppSource** structure (the second parameter to jseGetSourceFunc) has this prototype:

```
struct jseToolkitAppSource
  {
   char * code;
   cost char * name
   uint lineNumber
   void * userdata
  }
```

code contains the line of code read from the file. You must allocate memory for this pointer.

name is the name of the file being accessed.

lineNumber: This is the line number of the line that is to be read and stored in **code**. Both the application and core can read and write to this value. It is initialized to zero for each file and incremented before each call to jseGetSourceFunc(), so you don't need to modify this value if each subsequent call represents the next higher line number. This value is used for error reporting and debugging by the core.

userdata may be used to store any data you want to pass to the function.

flags: this will be one of the following values, depending on how the file is to be treated: **jseNewOpen**, **jseGetNext** or **jseClose**.

jseNewOpen indicates that the interpreter wishes to open the file;

jseGetNext reads and returns the next line of the file,

jseClose is sent when the file is ready to be closed.

Other Optional Functions

Set these options to NULL if not wanted.

jseMayIContinueFunc will be called after each command in the script is executed. If the continue routine returns *True*, the macro script will continue; if *False* is returned, the script will be terminated. If you do not want to have any function called, set this parameter to *NULL*. You can use this function to provide a debugging interface (use jseLocateSource() to retrieve the name and line number of the current location in the script being run), a callback monitoring function for your scripts, call multitasking tickler routines, or check on external status such as the pressing of ctrl-C or break.

```
typedef jsebool (JSE_CFUNC FAR_CALL *jseMayIContinueFunc)
                (jseContext jsecontext);
```

jseAtErrorFunc and Error Handling

An error can be generated in a number of ways: a code error can happen, like trying to read an undefined variable; an API wrapper function can use jseLibErrorPrintf to generate an error; the script can have a 'throw' statement.

In any case, your 'jseAtErrorFunc' is immediately called. A structure is passed to it that has information about the error. (This structure can be extended in future releases.)

> The first member is '**errorVariable**'. This is the error being generated. It is possible for this not to be an object, but it usually is. For instance, 'throw "foo";' will have the error variable be the string "foo". All error objects can be transformed into strings using 'jseCreateConvertedVariable(...,jseToString)'.
>
> The second parameter, **trapped**, determines whether or not the error will be trapped. Simply, if it is not trapped, your PrintError function (see below) will be called immediately after the AtError function. It means that no 'try/catch' handlers are in effect that could possibly catch the error.

At any rate, you are free to examine the state of the interpreter, check the line number, read variables, and so forth. If you are in a debugger, for instance, you can check the kind of error object to determine if you want to call a halt for the user, probably based on preferences the user has set.

The **ErrorMessageFunc** is called to deliver an error message to the user. Usually, you will print this to the screen or pop up a dialog box with the information. You can check variables, and so forth. However, note that this function is not called until the error is no longer trapped. For instance, if you are in a try block, then no error message will be printed at the point of the error, instead the code will unwind and the 'catch' handler gets to decide what to do with the message.

In a simple script, that uses no try/catch handling, the error function will print at the point of the error. In this case, the AtError func will be called first, if it exists. 'trapped' in the AtErrorStruct will be False. When this function returns, the ErrorMessageFunc will be called immediately.

```
   struct AtErrorStruct
   {
        jseVariable errorVariable;
        jsebool trapped;
   };

   typedef void (JSE_CFUNC FAR_CALL *jseErrorMessageFunc)\
                (jseContext jsecontext,const jsecharptr
ErrorString);
   typedef void (JSE_CFUNC FAR_CALL *jseAtErrorFunc)\
                (jseContext jsecontext,jseVariable errorObject);
```

AppLinkFunction is only used if you call jseAppExternalLinkRequest(); otherwise it should be set to *NULL*. jseAppExternalLinkRequest() is used to start up a new scripting session and create a new jseContext based on the current context. The AppLinkFunction initializes values for the new jseContext created.

```
typedef jseContext (JSE_CFUNC FAR_CALL *jseAppLinkFunc)
                   (jseContext, jsecontext, boolean,
                    initialize);
```

initialize is a boolean value indicating whether the script is being initialized or terminated. If it is *True*, the script is being initialized, the jseContext passed in is the context for the current (primary) session; the AppLinkFunction must return the context of the new session. If it is *False*, the script is terminating, and the jseContext passed in must be the jseContext returned from the call to initialize the session.

jseSecureCode is either a block of JavaScript code or a full file name and path to a script that performs the security checking. The format of such a file is explained in the next section. If there is no security check, this parameter must be set to *NULL*. This parameter is ignored by the 16-bit DOS version of the interpreter. (Security is not enabled in the 16-bit DOS version; please contact Nombas if you need security capabilities for 16-bit DOS).

options is an Or mask of the following values. These options configure how the interpreter handles variables:

jseDefault - Use this flag to use the system defaults.

jseOptRequireVarKeyword - Use this flag if you want to force your users to use the 'var' keyword when creating variables.

jseOptRequireFunctionKeyword - Use this flag if you want to force your users to use the 'function' keyword when creating functions.

jseOptDefaultLocalVars - Use this flag if you want variables declared in a local environment to be local, regardless of whether the var keyword is used or not. (In JavaScript, variables declared without the var keyword would normally be global). If there is a like-named global variable, instead of creating a local variable the global variable would be used.

jseOptDefaultCBehavior - If this flag is defined, functions will be treated as if they were created with the 'cfunction' keyword, regardless of what keyword they were defined with.

jseOptWarnBadMath - If this flag is set, the interpreter will notify you when you make illegal mathematical calculations (such as dividing by zero). In JavaScript, dividing by zero normally returns the value *NaN* and does not generate an error.

jseOptLenientConversion - this option causes variables to automatically be converted to the required type if possible, instead of generating an error. With this option set the macro JSE_VN_CONVERT() will always behave as if the first parameter passed were JSE_VN_ALL. The jsePutxxx() functions will convert the variable to the required type. If you are retrieving data from a variable that is not of the correct type, a copy of the variable will be made, converted to the correct type, and returned.

jseOptIgnoreExtraParameters - If this option is set, the interpreter will ignore any parameters greater than the maximum allowed for the function (specified in the Function Descriptor table added to the context with jseAddLibrary().

Terminating a jseContext

The jseContext must be terminated once you are done using it. This ensures that all system resources are properly freed. To terminate a jseContext, make a call to jseTerminateExternalLink(). This routine takes one argument, the root jseContext to terminate. This must be the same context returned by jseInitializeExternalLink(), and not one of the derived contexts. You must make a call to jseTerminateExternalLink() for each jseContext. After this call, no more calls may be made to the terminated context.

Terminating the interpreter engine

Make a call to jseTerminateEngine() to shutdown the ScriptEase Interpreter Engine. This call should be made only once, following the last call to jseTerminateExternalLink(), to free the resources allocated by the engine during the call to jseInitializeEngine(). You only need to make this call if you are linking with the static library version of the ScriptEase:ISDK/C. The dynamically linked version makes this call for you in its termination code.

Security code

You may provide a security filter to prevent certain functions from executing and to limit scripts to working in certain directories or files. The security filter is a script, so it can be easily modified without recompiling your application. The security script will be called before a script is run and before every function call. The full path and filename to the file containing the code is passed to jseInitializeExternalParameters() when obtaining the jseContext.

The security script contains a SecurityGuard() function, and optionally may contain SecurityInit() or SecurityTerm() functions. SecurityGuard() receives the name of the function being called as its second parameter and tests to see whether the function call is permitted or not. SecurityInit() is called before the script runs, and SecurityTerm() is called when the script terminates.

Here is an example of a security file for an application that wants to prevent users from using any unsafe functions except for fclose() and fopen(). (Unsafe functions are those that have the potential modify the files or operating system in some way). This security file limits the fopen() function to only work in certain directories, to prevent unauthorized snooping around, It also adds to the PATH variable before running the script, and removes it when it is done.

```
SecurityInit(SecurityVar)
{
 // initialize SecurityVar
 SecurityVar.TempDir = getenv("TEMP");
 AddPath(...add a few directories to PATH...);
 return True;
}
SecurityTerm(SecurityVar)
{
 DeletePath(...remove stuff added to PATH...);
 return True;
}
SecurityGuard(SecurityVar,FunctionName,var1,var2,var)
{
 switch( FunctionName )
 {
  case "FCLOSE":
     return True;   // always succeed
  case "FOPEN":
   // limit fopen()
  default:
  // no other security-risk functions allowed
  return False;
 }
}
```

Testing the integration

To be sure the interpreter has been integrated correctly, add a call to interpret a test script. The script can be simply "a=1", for example:

```
jseInterpret (jseContext, null, "a=1;", null, jseAllNew,
              JSE_INTERPRET_CALL_MAIN, null, &ReturnCode);
```

Add this function between where you call jseInitializeExternalLink and jseTerminateExternalLink. If you can compile, link, and run this test code, then the interpreter is properly included in your application and functioning correctly.

API error messages

The two functions jseGetLastAPIError() and jseClearAPIError() facilitate debugging the integration. If an API call is improperly made a message describing the error is set in the interpreter's error buffer. The error may or may not cause the interpreter to fail, although your application will most likely crash at some point if it uses the bad data. If a script causes your application to crash, see if an error message has been set by calling jseGetLastAPIError(). We recommend using frequent calls to jseGetLastAPIError() to catch these problems early on, particularly if you are debugging or are just starting out using the ISDK/C. jseClearAPIError() removes the current error message from the buffer.

Adding functions to the ScriptEase engine

Once you have initialized the interpreter and created a jseContext, you must register any functions you want to make available to your users. This is a three step process:

- Every function you make available to your users must be entered into a function library table. The function library table is an array of structures that contain the function's name as it will be called by your users, a pointer to the corresponding wrapper function, the minimum and maximum number of arguments for the function, and a mask of options.
- Call jseAddLibrary() to register the function library table(s) with the ScriptEase interpreter.
- Write wrapper routines for the functions you wish to add to the function library. The wrapper function retrieves the function arguments from the ScriptEase call, translates the data from ScriptEase to C, makes your application call, and then translates any C values back into ScriptEase for return.

Object methods and object constructor functions are included in your application in the same way as regular functions are. If the function is one of the Nombas supplied methods or standard ECMAScript/JavaScript objects, you don't need to worry about creating a wrapper routine. Instead, you can just define the method in the JSEOPT.H file. Instead of calling jseAddLibrary() to register the library with the jseContext, you must call the relevant LoadLibrary_xxx() function, as described in the appendix "Standard JavaScript Objects and Methods and Nombas' Extensions."

Creating a ScriptEase function library table

A ScriptEase function table is an array of function descriptors. Each element of the array specifies to the interpreter the details of a function to be added to the ScriptEase library. The function descriptor assigns the ScriptEase function its name, the address of the compiled C function, and the number of arguments (minimum and maximum) the ScriptEase function takes. There is no limit to the number of functions that can be specified in a function library table, nor is there any limit to the number of library tables that may be added to a given jseContext.

The ScriptEase:ISDK/C provides a number of macros to assist in building the table. Which macro you use will depend on the type of function being added to the table:

JSE_LIBOBJECT This defines what is being added to the table as an object. The object added with JSE_LIBOBJECT is the current object; it will remain current until another call to JSE_LIBOBJECT creates a new current object. Functions and variables added with the other macros (listed below) will be added as properties and methods of the current object.

JSE_LIBMETHOD To add a method or function to the current object. The method will be added to the last object called with JSE_LIBOBJECT.
JSE_PROTOMETH To add a method to the current object's prototype.
JSE_VARASSIGN This macro creates a copy of an already existing variable and assigns it to the current object.
JSE_VARNUMBER Use this macro to assign a numerical value as a property of the current object.
JSE_VARSTRING Use this macro to assign a string value as a property of the current object.
JSE_ATTRIBUTE This macro creates an undefined variable with specified attributes. It can also be used to change the attributes of an extant variable.
JSE_FUNC_END To indicate the last entry in a table

These macros have the following syntax:

```
JSE_LIBOBJECT(name, addr, min, max, varAttr, funcAttr)
JSE_LIBMETHOD(name, addr, min, max, varAttr, funcAttr)
JSE_PROTOMETH(name, addr, min, max, varAttr, funcAttr)
JSE_VARASSIGN(name, variable, varAttr)
JSE_VARNUMBER(name, var_number, varAttr)
JSE_VARSTRING(name, var_string, varAttr)
```

These macros take the following parameters, as indicated above:

name is a string representing the name to give to your function in a script. It should be an ASCII string such as "GetString". This is the name by which your users will refer to the added function or property.
addr is a pointer to the function called by the ScriptEase Interpreter Engine, i.e., the name of the wrapper function that corresponds to the function listed above.
min is used to specify the minimum number of arguments that can be passed to your new ScriptEase function (0-127).
max is used to specify the maximum number of arguments that can be passed to the function (0127). For no maximum, (i.e. variable argument lists) set MaxVariableCount to -1. Set the maximum and the minimum to the same value to specify an exact argument count. If a script calls a function whose parameters do not meet the function parameter requirements, the script will be terminated with appropriate error handling.

varAttr is an or mask of one or more of the following:

jseDefaultAttr is used as a place holder for this parameter when you don't want to use any of the other options.
jseDontEnum This will prevent the property or method from being listed.
jseDontDelete Don't allow deletes on this element
jseReadOnly Makes the property or method read only.
jseImplicitThis Add this to the prototype chain (functions only).

jseImplicitParents This option allows you to change the variable's scope chain, creating global variables from another object's variables. It works in tandem with the .__parent__ (two underscores preceding and following "parent") property. If a function variable has the jseImplicitParents option set, you can assign the properties of another object to it by setting the .__parent__ property of the object to which the function is assigned to the object whose properties you want to make available to the function. Consider the following example:

```
            var foo;
            foo.a = 0;

            var goo;
            goo.__parent__ = foo;
            goo.increment = my_increment;

        goo.increment();
        // The above call actually modifies foo.a, since goo's
        // parent is foo
        Clib.printf("foo.a = %d\n",foo.a);

        //(this function must have its jseImplicitParents flag
        // set)
        function my_increment()
        {
         a++;
        }
```

funcAttr is an or mask of one or more of the following:

jseFunc_Default to specify the default behavior
jseFunc_CBehavior specifies that the function uses C behavior regarding strings and variables
jseFunc_Secure indicates that the function is safe to call. If this is not supplied, a security risk is assumed.
variable is a variable which has already been included in the function library table.
var_number is a number variable or value to be assigned to the current object.
var_string is a string variable, enclosed in quotes, or a literal string enclosed in quotes ("\"literal string\"", e.g.), to be added as a property of the current object.

Initializing a ScriptEase function library table

A Function Library Table is initialized and added to a specific context by calling jseAddLibrary(). For example:

```
jseAddLibrary( jseContext, ParentObject, jseFunctionTable,
                  jseLibraryInitData, jseLibraryInitFunction,
                  jseLibraryTerminateFunction);
```

jseContext is the jseContext to which the library will be added. Use the value returned by jseInitializeExternalLink().

ParentObject is the object to which the function belongs. Set this to "Global" to make the function available to all objects.

jseFunctionTable is the array of ScriptEase function descriptors to be added to the library.

jseLibraryInitData is a variable to contain any data that must be passed to jseLibraryInitFunction (described below). If there is no data to pass, set to *NULL*.

jseLibraryInitFunction is a pointer to the function the interpreter engine will call once before using the library being added. The return value from the jseLibraryInitFunction is a generic pointer to the library data. This data is available to all library functions via jseLibraryData().

If the function being added to the interpreter is a constructor function for an object, the jseLibraryInitFunction may assign properties to the object with jseMember(), jseAssign(), jseMemberWrapperFunction(), or a related function.

For example, a structure could be allocated in the initialization function and returned as the library data. All functions within the library can make a call to GetLibraryData() to access the allocated structure. The prototype for jseLibraryInitFunction() is:

```
void _FAR_  __cdecl FAR_CALL
*jseLibraryInitFunction( jseContext jseContext, void_FAR_
                         *PreviousInstanceLibraryData );
```

jseContext is the context provided by the interpreter engine for use by ScriptEase:ISDK/C calls in your jseLibraryInitFunction().

PreviousInstanceLibraryData is a pointer to the library data for a previous instance of the same library if one exists. In most cases this is set to *NULL* (unless the library is being reinitialized in a recursive call to jseInterpret()).

jseLibraryTerminateFunction is the function you provide that will be called when the library is no longer in use by the interpreter engine. This is called following the a call to terminate a jseContext by jseTerminateExternalLink() (or when a recursive call to jseInterpret() terminates). When the library terminate function is called, it must free any data that was allocated in the initialize function. The jseLibraryTerminateFunction() prototype is:

```
void _FAR_ __CDECL FAR_CALL
jseLibraryTerminateFunction(jseContext jseContext,
                            void_FAR *InstanceLibraryData);
```

jseContext is the context provided by the ScriptEase Engine for use in ISDK/C calls in your jseLibraryTerminateFunction().

InstanceLibraryData is a pointer to the library data for this library. If the library data was allocated in the initialize library function, free it in the terminate function. Either the init or the term functions may be *NULL* if you do not require these features.

Writing ScriptEase function wrappers

Writing wrapper functions for new ScriptEase functions involves three steps:

- Retrieve the function arguments passed in the script. First you must create variables to contain the function arguments, and then extract the data from the script with one of the jseGetXXX() functions. This will translate them from ScriptEase to C for use in your code;
- Calculate the value to be returned and execute any functions required to execute the script's function;
- Call on of the jseReturnXXX() functions to return a value to the script.

Wrapper routines take the following form:

```
void JSE_CFUNC FAR_CALL FunctionName(jseContext jseContext)
```

The jseExternalLibFunc(FunctionName) macro can be used to prototype and define your wrapper routines.

Retrieving function arguments in a wrapper function

Three functions get a jseVariable pointer to function arguments: jseFuncVar(), jseFuncVarNeed(), and the C++ macro JSE_FUNC_VAR_NEED(). Which method you use will depend on the type of variable, your application environment, and programming style. The end result of each is the same: to provide a handle to a variable your code can use to store a value passed from a script.

jseFuncVarNeed()

jseFuncVarNeed() provides type checking on a jseVariable being passed from a script. The script being interpreted will generate an error message and terminate if the appropriate variable is not found. The variable's type is checked just prior to retrieving its handle, so the handle returned can be relied upon to be a valid jseVariable and of the type expected.

```
jseVariable JSE_CFUNC
jseFuncVarNeed(jseContext jseContext,
               uint ParameterOffset,
               jseVarNeeded need );
```

jseFuncVarNeed() takes three parameters: the jseContext for the current scripting session, the offset of the parameter in the parameter list (0 for the first parameter, 1 for the second, etc.), and a value indicating the argument type expected by the script's function.

This value may be one or more of the following, depending on the variable type you expect to receive:

```
JSE_VN_NUMBER
JSE_VN_STRING
JSE_VN_BOOLEAN
JSE_VN_BUFFER
JSE_VN_NULL
JSE_VN_OBJECT
JSE_VN_FUNCTION
JSE_VN_BYTE
JSE_VN_INT
JSE_VN_ANY
JSE_VN_NOT()
JSE_VN_CONVERT(from, to)
```

If a variable may have more than one possible type, all possible types should be supplied, joined by a bitwise or (JSE_VN_STRING | JSE_VN_NUMBER, e.g.). The last three values on the list are used when the variable type is unknown and in cases where more than one variable type is expected. JSE_VN_ANY will accept a variable of any type. JSE_VN_NOT() will accept a variable of any type other than those passed to the macro. If you are passing more than one value to JSE_VN_NOT, they should be joined by an or (|). JSE_VN_CONVERT() takes two parameters. The first is an or mask of variable types to be accepted; the macro converts the variable to whatever type is specified by the second parameter.

The jseContext used in these calls must be the context passed to your wrapper function. Once a jseVariable handle has been obtained, data can safely be extracted, set to a new value or converted to a new type.

jseFuncVarNeed() returns NULL on failure. If NULL is returned, your error routine will have been called, and the script being interpreted will abort when it returns from your wrapper function.

jseFuncVar()

If a function expects a variable of unknown type or if the wrapper function assigns a type to the variable, use jseFuncVar() to obtain the variable's handle. It retrieves a variable handle regardless of its type. jseFuncVar() is prototyped as:

```
jseVariable JSE_CFUNC
jseFuncVar( jseContext jseContext, uint ParameterOffset );
```

Like jseFuncVarNeed(), jseFuncVar() returns *NULL* on failure. If *NULL* is returned, your error routine will have been called, and the ScriptEase macro script being interpreted will abort on return from your wrapper function.

JSE_FUNC_VAR_NEED()

The JSE_FUNC_VAR_NEED() macro may be used with C++ compilers to simplify the retrieval of validated variables from the parameters passed to a library function and testing them for success. Since JSE_FUNC_VAR_NEED() returns from the wrapper function if the variable is not valid, all statements following JSE_FUNC_VAR_NEED() can safely assume valid variables.

The JSE_FUNC_VAR_NEED() macro is defined as:

```
JSE_FUNC_VAR_NEED(varname,jsecontext,ParameterOffset,need)
   jseVariable varname;
   if ( NULL == ( varname = jseFuncVarNeed (jseContext,
        ParameterOffset, need)) )
   return
```

JSE_FUNC_VAR_NEED() will return if an error occurred checking the variable, so this function is most useful in the beginning of your function, before any statements that may need cleaning up prior to returning from the function. For instance, you would not want JSE_FUNC_VAR_NEED() after fopen() because it could return without calling fclose().

Getting Data From jseVariables

Once you have a jseVariable handle, the data can be retrieved by calling the appropriate jseGetXXX() function. There is a specific get function for each of the ScriptEase types (jseGetLong() and jseGetString(), e.g.).

For example, if your function had one argument that was a number, you would use jseGetLong() to get the value of the ScriptEase variable.

```
// get the value from a jseLongVar.
longArgumentVal = jseGetLong(jseContext, jseLongVar);
printf("%d", longArgumentVal);
```

If you used jseFuncVar() to get the handle to a function argument, first check the ScriptEase type before accessing its data. jseGetType() will return a jseDataType of jseTypeByte, jseTypeLong, jseTypeFloat, jseTypeObject or jseTypeUnknown. If you try to access the data of a jseVariable of one type with the jseGetXXX() function of another type, the interpreter will attempt to convert the data type from the actual type to the type requested if those types are convertible.

You do not need to check the type if you are just assigning a value to the variable and not extracting its value, although you may need to convert its type with jseConvert().

Assigning values to jseVariables

Setting the value of a jseVariable is similar to getting the value. Using the jseVariable handle, call one of the jsePutXXX() functions. For example, if your function had one argument that was an integer, you would use jsePutLong() to set the value of the ScriptEase variable.

```
longValue = 1000 * 1000;
jsePutLong(jseContext, jseLongVar, longValue);
```

To ensure that the new value is of the appropriate data type as the jseVariable, use jseConvert() before assigning the new value. jseConvert() will return with an error if it cannot convert the variable to the requested type.

The jsePutXXX() functions will have a permanent effect only if the wrapper function is for a cfunction or an object.

Returning values from a wrapper function

To return a primitive value from a wrapper function, use the appropriate jseReturnXXX() function: to return a long, use jseReturnLong(); to return a float, use jseReturnNumber(). Both require the context that was passed to your wrapper function as the first argument, and the value to return as the second.

```
// Return a long from a ScriptEase wrapper function.
jseReturnLong(jseContext, 3006);

// Return a float from a ScriptEase wrapper function.
jseReturnNumber(jsecontext,22.22);
```

Returning an object requires a call to jseReturnVar(). jseReturnVar() puts a generic data type on the jseStack to be returned to the script. jseReturnVar() can be used to return data of any type, although it is better to use the typed functions (jseReturnLong(), e.g.) if possible. Call jseReturnVar() with three arguments:

```
jseReturnVar(jseContext, MyjseVar, jseRetTempVar);
```

The first argument, jseContext, is the jseContext provided by the wrapper function. The second argument is the jseVariable to return. The last argument is the return action. This argument tells the ScriptEase engine what to do with the data space once the function has returned and the statement that called the function has completed. Possible values for this parameter are:

jseRetTempVar - the variable will be destroyed when popped from the stack.

jseRetCopyToTempVar - the variable is copied, and the copy is put on the stack. It will be destroyed when popped from the stack.

jseRetKeepLVar - the variable will not be destroyed unless instructed to do so with a call to jseDestroyVariable.

In nearly all cases this should be set to jseRetTempVar.

Passing and returning simple data types

Passing and returning one of the primitive data types (numbers, strings and booleans) involves calling jseFuncVarNeed() to get the appropriately typed jseVariable and then calling jseGetLong (if a number has been returned, for example) to extract its value. The

following example is a wrapper for a function that simply adds two integers and returns the result. It would be invoked from the script source like this:

```
sum = MySumFunction(var1, var2);
```

Here is the wrapper function:

```
ExternalLibFunc(MySumFunction) {
   jseVariable MyjseInit1;
   jseVariable MyjseInit1;
   long MyCint1;
   long MyCint2;
   long MyCint3;
   long MySumInt;

MyjseInt1 = jseFuncVarNeed(jseContext, 0, JSE_VN_NUMBER);
MyjseInt2 = jseFuncVarNeed(jseContext, 1, JSE_VN_NUMBER);
if(MyjseInt1 == NULL || MyjseInt2 == NULL) {
 return;
}
MyCint1 = jseGetLong(jseContext, MyjseInt1);
MyCint2 = jseGetLong(jseContext, MyjseInt2);

MySumInt = MyCint1 + MyCint2;

jseReturnLong(jseContext, MySumInt);
return;
}
```

If there is not enough memory to complete the call or an invalid parameter is passed in, jseFuncVarNeed() will return *NULL* and call the user-defined error function. The interpreter will not quit until the wrapper function ends, so it is imperative to exit the function before it tries to use the bad data. This is why calls to jseFuncVarNeed() are put at the beginning of the function.

If the call to jseFuncVarNeed() is successful, it returns a pointer to the space in the interpreter's memory that holds the value of var1. Call jseGetLong() to extract the value from the variable. Do not access its value directly.

The ISDK/C makes no distinction between short and long integers. Both sizes of integer use jseGetLong(). Bytes and floats returned by jseGetLong() will be cast as long integers.

Returning an integer requires just one function call, jseReturnLong(). This function assigns the C variable's value to the jseContext. The ScriptEase interpreter will internally allocate the data space needed to hold the value.

Passing and returning strings and boolean values is essentially the same. Strings and booleans both have their own type parameters to the jseFuncVarNeed() call: use JSE_VN_STRING for strings and JSE_VN_BOOLEAN for boolean values. Use jseGetString() to extract data from strings, and jseGetNumber() to extract data from booleans; use these functions in place of jseGetLong() in the example. To return strings or booleans use jseReturnVar().

For example, here is a script that passes and returns a string:

```
jseExternalLibFunc(PromptAndGetS) {
 jseVariable MyjseBuffer;
 char szTextBuffer[80];

 MyjseTextBuffer = jseFuncVarNeed(jseContext, 0,
JSE_VN_STRING);
 ulong str_len;
 strncpy(szTextBuffer, (char*)jseGetString(jseContext,
        MyjseBuffer, &str_len), 78)[79] = '\0';
 printf("%s", szTextBuffer);
 gets(szTextBuffer);
 jsePutString(jseContext, MyjseBuffer, szTextBuffer,
             strlen(szTextBuffer) );
 jseReturnVar(jseContext, MyjseBuffer, jseRetCopyToTempVar);
 return;
}
```

Passing simple data types by reference

In addition to return values, your wrapper function can return data directly via the variables on the jseStack. For example, suppose you had a ScriptEase function that modified a number in some way:

```
num = 10;
ModifyNumber(num);
if( num == 0 ) exit(EXIT_ERROR);
```

To make this function available to your users you must create a wrapper function such as the following:

```
ExternalLibFunc(ModifyNumber) {
 jseVariable   MyjseL;
 MyjseL = jseFuncVar(jseContext, 0);

 // If returned NULL this type can't convert to an integer
 if( NULL != jseConvert(jseContext,MyjseL,jseTypeLong, 0))
 {
  jsePutLong(jseContext, MyjseL, GetANumber());
 }
 return;
}
```

When you pass simple data types by reference you can skip the type checking of the jseVariable. The current variable type is of no importance (it may not even have a type yet (jseTypeUnknown)) because we will use jseConvert() to ensure that the variable is of the correct type.

You still need to associate the parameter offset with a jseVariable. Call jseFuncVar() with the parameter offset as the second argument to get a jseVariable. Now, convert the jseVariable into one that will hold a long using jseConvert(). jseConvert() requires the jseContext, the jseVariable to convert and the new variable dimension. In our case, we will convert num to a single jseTypeLong. Finally, the value returned by GetANumber() is passed to jsePutLong(). Calling jsePutLong() inserts the C variable's value into the

jseVariable. Upon returning to the script, the script's variable will hold the value returned by the function GetANumber().

Working with objects

Passing and returning objects involves an additional step. As with the primitive data types, you must first get a jseVariable for the object with jseFuncVarNeed(). Then get a handle to the property by calling jseMember(). The data may now be extracted from this second jseVariable with a call to one of the jseGetXXX functions.

jseMember() has four parameters: the relevant jseContext, the name of the object whose members are being accessed, the name of the property being accessed, and the data type of the property.

```
     jseExternalLibFunc(jseObjectFunc) {
      jseVariable   jseVarObject;
      jseVariable   jseVar;
      int           integer = 0;
      char          string[255];
      string[0] = '\0';

      jseVarObject = jseFuncVarNeed( jseContext, 0,
   JSE_VN_OBJECT);
      if (jseVarObject == NULL) return;
      jseVar = jseMember(jseContext, jseVarObject, "MyInt",
                         jseTypeNumber );
      if (jseVar != NULL) {
       integer = (int)jseGetLong(jseContext, jseVar);
      }
      jseVar = jseMember(jseContext, jseVarObject, "MyString",
                         jseTypeString);
      if (jseVar != NULL) {
       ulong str_len;
       strcpy(string,jseGetString(jseContext, jseVar, &str_len));
      }
      printf("string=%s, integer=%d\n", string, integer );
      jseVar = jseMember(jseContext, jseVarObject, "MyNumber",
   jseTypeNumber );
      if (jseVar != NULL) {
       jsePutLong(jseContext, jseVar, 17);
      }
   return;
```

The example above gets an object with two properties from the interpreter. One of the properties (MyString) is a string, and one of them (MyInt) is a number; they will be stored in the variables integer and string, respectively, and printed to the screen.

The script then tries to get a handle to a MyNumber property. Since no such property exists, the call to jseMember returns *NULL*, and creates the new object property. A value (17 in this example) is then assigned to the property with a call to jsePutLong().

Since jseMember() doesn't allocate any memory in creating new object properties, you shouldn't try to destroy them when you are through with them. Child variables will be cleaned up by the interpreter engine when the parent object is destroyed.

Functions with a variable number of arguments

ScriptEase functions may accept a variable number of arguments. For example, consider the following function, which takes a string as its first parameter, and has an optional second parameter, a number:

```
ret = OneOrTwoArgs("My Dog Has Fleas");       // returns 'M'
ret = OneOrTwoArgs("My Dog Has Fleas", 7);    // returns 'H'
```

If the integer parameter is not provided, the function returns the first character of the string. If an integer is provided, the character returned will be at the index position specified by the integer.

Since the first argument is mandatory, there is no need to treat it differently. It may be accessed just as in the previous examples. However, you must determine whether the second parameter exists before you try to extract a value from it. The function jseFuncVarCount() will return the number of parameters specified for the ScriptEase function. If the variable exists, the usual jseFuncVarNeed() and jseGetLong() calls to check the jseVariable type and extract its data.

```
jseExternalLibFunc(OneOrTwoArgs) {
 jseVariable  MyjseString;
 jseVariable  MyjseOptNum;
 char        *MyCstr;
 ulong        MyCoptNumber;
 char         MyCchar;
 int          index;
 MyjseString = jseFuncVarNeed(jseContext, 0, JSE_VN_STRING);
 if (MyjseString == NULL) {
  return;
 }
 ulong str_len;
 MyCstr = (char *)jseGetString(jseContext,
                               MyjseString,&str_len);
 MyjseOptNum = 0;
 if (jseFuncVarCount(jseContext) == 2) {
  MyjseOptNum = jseFuncVarNeed(jseContext, 1, JSE_VN_NUMBER);
  MyCoptNumber = jseGetLong(jseContext, MyjseOptNum);
 }
 if (MyjseOptNum<strlen(MyCstr)) index = MyjseOptNum;
  else index = 0;
  MyCchar = MyCstring[index];
  jseReturnLong(jsecontext,MyCchar);
  return;
 }
```

Accepting a ScriptEase argument of unknown type

If you do not know what type of variable is to be retrieved, you can use the function jseFuncVar() instead of jseFuncVarNeed(). jseFuncVar() will accept a variable of any type. You must then call jseGetType() to determine the variable's type. jseGetType() returns the jseDataType, which will be one of the following: jseTypeUnknown, jseTypeObject, jseTypeByte, jseTypeLong, or jseTypeFloat. This provides you with the information you need to execute the proper C code.

The following example shows the jseVariable jseMysteryVar first being assigned a handle into the interpreter engine using jseFuncVar(). Before this variable can be used, its ScriptEase type must be determined with a call to jseGetType().

```
jseExternalLibFunc(AnyVarArgs)
{
 jseVariable  jseMysteryVar;
 char         *MyCchar;
 int          MyCNumber;
 uchar        MyCbool;
 ulong        len;

 jseMysteryVar = jseFuncVar(jseContext, 0);
 switch(jseGetType(jseContext, jseMysteryVar)) {
  case jseTypeUndefined:
   /* Can set to a jseType here */
   jseConvert(jseContext, jseMysteryVar, jseTypeNumber);
   break;
  case jseTypeString:
  case jseTypeBuffer:
   MyCchar = jseGetBuffer(jseContext, jseMysteryVar, &len);
   break;
  case jseTypeLong:
   MyCNumber = jseGetNumber(jseContext, jseMysteryVar);
   break;
  case jseTypeBoolean:
   MyCbool = (uchar) jseGetNumber(jseContext, jseMysteryVar);
   break;
 }
 return;
}
```

This function executes different code depending on the variable type passed. The correct data extraction function will be called against the ScriptEase variable no matter what the data type is.

Calling interpreted ScriptEase functions

Once a function has been interpreted with jseInterpret(), it has been registered with the interpreter. To call the function again you can call jseCallFunction(); this saves the interpreter from having to re-interpret the function. You can use jseGetNextFunction() to list all available local functions in a given jseContext.

There are five steps to calling previously interpreted (via jseInterpret()) functions:

- get a handle to the function with jseGetFunction()
- Create a jseStack to manage the variables used by the function
- put variables on the stack with jsePush()
- make the function call with jseCallFunction()
- destroy the jseStack with jseDestroyStack()

jseGetFunction() returns a handle to the function, which will be needed for the call to jseCallFunction(). You may also use the jseMemberFunctions and jseIsFunction to get a variable that can be called as a ScriptEase function.

Creating a jseStack is easily done by calling jseCreateStack(). This function creates the stack and allocates the memory it needs. It returns a handle to the stack, which is used in subsequent calls to jsePush() and jseCallFunction().

Next use jsePush() to put jseVariables on the stack. You must call jsePush() once for each argument you are passing to the function. jsePush() takes four parameters: jseContext, the handle of the stack you're working on (as returned by the call to jseStack()), the variable to be pushed to the stack, and a boolean flag. Set this flag to *True* if you want the jseVariable to be automatically destroyed when the stack is destroyed (after the function returns).

Now you are ready to make the function call with jseCallFunction(). jseCallFunction() returns *True* if the function was successfully executed, otherwise it returns *False*.

Creating (and destroying) jseVariables

ScriptEase variables are created using these jseCreateXXX() functions, each of which returns the created variable:

```
jseCreateVariable()
jseCreateSiblingVariable()
jseCreateLongVariable()
jseCreateConvertedVariable()
```

Any time a variable is created with any of the above functions, it must eventually be destroyed. There are two ways to destroy a ScriptEase variable:

- jseDestroyVariable() will destroy any variable created with the above calls.
- If RetAction is jseRetTempVar, jseReturnVar() will destroy the ScriptEase Variable after it is used. Do not destroy the variable explicitly if it is used as a return variable in this manner.

Interpreting a ScriptEase script

Once the interpreter has been initialized, your libraries have been added to the ScriptEase library and any global jseVariables have been established, the ScriptEase engine is ready to interpret a script. A script can be interpreted at virtually any time during the execution of your application after you have initialized the interpreter and functions as described above. Calling jseInterpret() begins the interpreter.

jseInterpret() returns a boolean value to indicate the success or failure of the interpretation process. jseInterpret() returns *True* if the script executed completely, otherwise it returns *False*. This value is in no way related to the return value of the script interpreted.

jseInterpret() - flags

These are the three most commons situations encountered when executing a script:

1. You want to execute the code as if it were the only thing running; all variables created will be destroyed.
2. You want your code to be able to use all variables that are currently available for the jseContext, and all variables created by the script will remain after the script terminates.
3. You want your code to be able to use all variables that are currently available for the jseContext, but you don't want the variables created by the script to remain after the script terminates.

The flags to use for the **jseNewContextSettings** and **howToInterpret** for these three situations are as follows:

1. jseNewContextSettings:jseAllNew & ~jseNewSecurity
   - howToInterpret: JSE_INTERPRET_CALL_MAIN |
   - JSE_INTERPRET_NO_INHERIT
2. jseNewContextSettings: jseNewNone
   - howToInterpret: JSE_INTERPRET_CALL_MAIN
3. jseNewContextSettings: jseNewFunctions
   - howToInterpret: JSE_INTERPRET_CALL_MAIN

jseInterpInit(), jseInterpExec(), and jseInterpTerm()

These three functions provide an alternate method for interpreting scripts. jseInterpret() reads in a script, and then executes it statement by statement, calling jseMayIContinueFunc after executing each statement. If you want greater control over this procedure, you may use the three jseInterpXXX() functions.

jseInterpInit() complies the script into the interpreter. It takes the same parameters as jseInterpret(). It does no script execution; it just initializes the script. jseInterpExec() executes the next line of a script, and jseInterpTerm() is called when the script is to stop execution. These latter two functions are only passed the jseContext.

Interpreting in pieces:

Once you have set up for an interpret session using jseInterpInit(), you actually execute the ScriptEase statements using successive calls to jseInterpExec(). Initially, you pass the context you received as a result of jseInterpInit(). You will be returned a new context that

you pass back to execute the next statement. Continue calling this function with the result of the last call as the parameter.

When this function returns NULL, the interpret is complete. At this time, you call the jseInterpTerm() function to clean everything up and get the return value.

If you call jseInterpret(), your 'MayIContinue' function is called after each statement. If you use the jseInterpXXX() functions directly, your 'MayIContinue' statement is NOT called. Instead, you may execute whatever code you like between successive calls to this function. You may decide to discontinue executing code by calling jseInterpTerm() at any time. If you do so, no return value can be given (the function always returns NULL).

Certain functions, due to the design of the interpreter, must be completely processed and so are atomic to this function. This means that if you do a jseCallFunction() or access a dynamic object, the statement will not be executed iteratively by your jseInterpExec() but will be handled behind the scenes all at once. Thus, when you call jseInterpExec() in these cases, many statements can be processed for your one call. In this case, your MayIContinue() function WILL be called during these statements. Thus, you should always have a valid MayIContinue() function.

jseInterpInit returns a new jsecontext. If it returns NULL, some error happened (like a syntax error during parsing.) The error message will have already been printed. You can trap this error by using the JSE_INTERPRET_TRAP_ERROR flag in the howToInterpret settings. The returnVar parameter is provided for this case. It will be filled in with the trapped error object. It is only used in this case; you can ignore it if you are not trapping errors. If you do trap an error (i.e. it is not NULL, it must be destroyed when you are done.)

You can trap any errors in the call to jseInterpTerm(), there being a boolean flag to do so. If you don't, error messages will be printed and a NULL will be returned on error. Otherwise, again, no message is printed and instead the error object is returned. You can use jseQuitFlagged() before calling jseInterpTerm() to determine if the result will be an error. If you are canceling the interpret as described above, this does not apply.

Exception Handling Via the API

In JavaScript, each function returns a value. This is done with the familiar 'return' statement. If you 'return 10;', the result of the function is 10. What you might not know is that actually the result is 10, plus an indication that no error occurred. Most JavaScript users think that scripts can only return values, and that any errors immediately abort the script. That is only an approximation that holds true most of the time. If you want to use the advanced capabilities of the JavaScript language, you'll need to know how things are actually happening under the hood.

Most errors are generated internally by the engine, when bad code is executed. For instance, if you try to access an undefined variable, an error occurs. JavaScript provides a little-used 'throw' statement to raise your own errors. In either case, this is treated very much like a 'return' statement, except that the value being returned is an error object (see below for details of these) and the indication is that an error did occur. If the function has a try/catch handler, it handles the error immediately instead of returning it. See the manual chapter on statements for details on try/catch.

If there is no such handler, then the error is returned to its calling function, who then does the exact same thing (either handling it or returning it to its caller.) Eventually, either someone will have handled the error, or the top-level of the script is reached. If it gets all the way to the top, and no one has handled the error, then the ScriptEase engine calls the application's error handling routine (defined in the jseExternalLinkParameters). For most applications, this just prints the error to the screen.

So, we can see that if no try/catch handlers are in place, then the behavior is what most users expect, the script stops running and an error message is printed. If, however, try/catch handlers are used, they can trap the error. In addition, the idea that an error is just a normal return plus an indicator that this is in fact an error is important to the ScriptEase API user, so keep it in mind below when we discuss the API.

What Is an Error?

Now that you know that an error is just a value returned that has the extra indication that it is an error, what is the value itself? Well, in an object-oriented language like JavaScript, it is not hard to guess that it is an object. In fact it is one of a number of related types of objects defined by the ECMAScript version three draft. We have SyntaxError, TypeError, EvalError, and several others. There is also a more generic Error object. If you want to use the 'throw' statement to generate an error, you will generate one of these kinds of errors. For instance,

```
throw new Error("13: This is not your lucky day!");
```

As was stated before, whenever a program error occurs, such as referencing an undefined variable, the engine constructs the appropriate error object and 'throw's it itself. These error objects, when converted to a string, all give a human-readable message that the programmer can understand. For program-generated errors, the message includes the line number and filename of the error.

Now, the tricky part is that you can 'throw' any value. You just normally throw one of the error objects, because that is the accepted practice, and it is what other people will expect you to do. Assuming your program does not trap the error, whatever value you throw will be converted to a string and printed out as an error message. The user can only recognize that an error occurs if the message makes it clear. Error objects get converted to a very descriptive string that is obviously an error message. If you start throwing various values that are not error objects, you'll make your users really confused. These will be converted

to a string and printed. The user may not even realize that it is an error! If you 'throw 10;', '10' will be printed as the error message. Your user will be confused as to what is going on. You are much better off always throwing error objects, not arbitrary values.

Creating Errors Via the API

Now that we understand errors from the script's perspective, what about from the API user's perspective? When you are writing wrapper functions, you are used to using the ScriptEase API call jseReturnVar() to return a value for that function. This is exactly like using the script 'return' statement. If you want to generate an error, to mimic the script's 'throw' statement, you can also use jseReturnVar(). You call jseLibSetErrorFlag() to indicate that this is an error return (i.e. a 'throw', not a 'return'.) However, doing this means you must build an appropriate error object to return. Creating a new object from the API can be a bit of a pain. It involves finding the constructor function you want to call, setting up the stack with its parameters, calling the function and finally destroying the stack.

Because this is such a common occurrence, we have a much simpler way to do it. You use the jseLibErrorPrintf() routine. This one function wraps the functionality of both jseReturnVar() and jseLibSetErrorFlag() into itself. You can pass a simple text string to it, and it turns that into an appropriate error object and makes it the return of your wrapper function. It also notifies the engine that an error has occurred, so after you call jseLibErrorPrintf(), your can just return from your wrapper function. The error will be passed back up the chain just like any other error, being trapped by a try/catch handler, or printed out to the user is no handlers exist.

When using jseLibErrorPrintf, it is preferable to use the ScriptEase resource capabilities to generate the error message, since then you do not have to worry about the format of the error string. All of the standard library functions we provide have their error messages generated this way, such as the standard ECMA functions in srclib/ecma/*.c. The file srclib/common/setxtlib.h contains the resource definitions they use.

If you need to generate an error message by hand, the format of the string passed to jseLibErrorPrintf is:

```
!type number: message
```

For instance,

```
!SyntaxError 1000: You did something wrong.
```

This functions just like the ScriptEase statement,

```
throw new SyntaxError("1000: You did something wrong.");
```

Remember that after you call jseLibErrorPrintf(), it has the return value all set up to give the engine the correct error. Make sure you do not then change it by calling jseReturnVar(). You should usually immediately 'return' from your wrapper function right after calling jseLibErrorPrintf. A few API function specify that if some parameters

to them are wrong, an appropriate error message will be generated. For instance, if you call jseFuncVar() trying to get a parameter out of range, then an error message will be generated. Exactly the same thing applies, if these functions fail, you should immediately return from your wrapper function. In case your wrapper function is complex, you can check to make sure no error has been generated before you try to return a regular value from it. Here is a short code example:

```
if( !jseQuitFlagged(jsecontext) )

jseReturnVar(jsecontext,my_return_variable,jseRetTempVar
);
else
   jseDestroyVariable(jsecontext,my_return_variable);
```

Notice the 'else' clause in this example. In most cases you will be returning a variable using the 'jseRetTempVar' option, which means that it is a variable you must destroy. By returning it, you are effectively destroying it. If you do not return it, you will have to then explicitly destroy it, as we have done in this example.

Catching and Propagating Errors

The ScriptEase API has two basic functions to actually execute code, jseInterpret() and jseCallFunctionEx(). jseCallFunction() and jseInterpInit() et al are variations of these two basic functions. Both of these represent the 'top-level' we talked about earlier when discussing error handling. If the error reaches them, they usually print out the error and return a boolean False to indicate that the function failed. They only return a result (by filling in the return variable parameter) if the function succeeded. You can, however, trap the errors with these calls. In this case, the return variable parameter is always filled in. You still know whether that value is a normal successful return or an error return based on the boolean return value of these functions. Read the API reference chapter for full information on their parameters and returns.

Certain wrapper functions will use this functionality to execute a script and return its result, even if it is an error. The ECMAScript eval() function (which we implement in 'srclib/ecma/ecmamisc.c') is a prime example. In this case, we simply set our API call to trap the error, and return the result whatever it may be. If the call resulted in an error, we also use the jseLibSetErrorFlag() API call to notify the engine of that.

Here is a simply wrapper function that demonstrates how an error can be passed along:

```
static jseLibFunc(example_wrapper)
{
  jseVariable retvar;
  jsebool waserror;

  waserror = !jseInterpret(jsecontext,NULL,
                           "var a = = 4;",
                           /* this will cause a syntax
error */
                           jseNewNone,
                           JSE_INTERPRET_TRAP_ERRORS,
                           NULL,
                           &retvar);
  jseReturnVar(jsecontext,retvar,jseRetTempVar);
  if( waserror ) jseLibSetErrorFlag(jsecontext);
}
```

Debugging

Once you get the ScriptEase ISDK integrated into your application, you can still have problems. The most common is misusing the API, for example destroying a jseVariable you were not supposed to. It is easy to make this and similar mistakes, so we've included a number of debugging tools to make getting your application up and running with the ScriptEase ISDK easier.

You'll need to turn debug support on while developing your application to get all of these benefits. Remember to turn it all back off when everything is working. The ISDK in debug mode is significantly slower and uses lots more memory to do all of its debug tracking. Make sure that the macro 'NDEBUG' is not defined. This macro is the key to determine which 'mode' (debug or release) the ScriptEase code is built in.

You'll also want to set your compiler to build a debug executable, to put full debugging information into your executable. Note that the debug changes to the ScriptEase ISDK core are compile-time, which means you'll need to recompile all ScriptEase files with these new changes, including the ISDK core itself. We include debug versions of the target files to compile the core, which you can use to build an appropriate build.

The first thing to note is that there are a lot of assert()s in the core, which are active in the debug mode. If you trigger an assertion failure, you've found some problem, either misusing the API, or a real bug in our code. In either case, the location of the assertion and what you did to cause it are important pieces of information which we can use to track down the problem.

JSEDEBUG.LOG

This is the name of the debug output file under DOS, Windows, or OS/2. For UNIX and Mac versions, it is put in the current directory. The debug output will be put in this file,

appended to whatever the file already contains. When trying to debug your program, delete this file first, then run it, then read it to see what information it provides. Even if this information is not enough for you to fix the problem, it will be helpful to us, so include it when you contact Nombas for technical support.

JSE_TRACKVARS

The most common mistakes involving the ScriptEase API have to do with incorrectly using jseVariables. When debugging mode is on, the macro JSE_TRACKVARS is defined, which causes the core to keep very careful track of all jseVariables created or looked up via API calls. This uses a significant amount of memory, but it is only meant for debugging.

The benefit is that most bad uses of variables will be caught and immediately flagged. If you pass a pointer to a jseVariable that is bad, or try to free a jseVariable you shouldn't, or try to use a jseVariable you have already freed, this will immediately catch the mistake and inform you of it. The majority of developer problems are of the kind that this setting will catch. You will save a lot of time by making sure that the JSE_TRACKVARS debug code does not catch any mistakes before you contact Nombas for support.

Memory Tracking

The second main benefit of the ScriptEase ISDK debug code is that it internally tracks all memory allocated. When memory is allocated or freed, it is filled with garbage values, so memory that is freed but later accessed will contain garbage, and thus be likely to cause an immediate problem. If your program crashes, look at the data structure involved. If it is working with a dynamically-allocated memory and has the hex value '0xEE' (for instance, the pointer 0xEEEEEEEE), it is likely an uninitialized value, since all allocated memory is filled with this value. If you instead find '0xBD' (or 0xBDBDBDBD as a pointer), you are using memory that has since been freed. Of course, you'll then need to track down why you are using bad memory, but at least you know that you are.

Second, on exit, all memory is examined, and if some memory was not freed, it will be reported (in the 'jsedebug.log' file, described above.) Special markers are written before and after each memory block, so if you have gone outside the bounds of the allocated memory, this too will be caught. These reports tell you the file name and line number that the memory was allocated on.

JSEMEMREPORT

Although pointer-bounds problems will be caught on exit, many times this mistake will cause the program to crash long before exit. If you suspect the problem may be due to memory corruption, put the call 'jseMemReport(False);' in your code before the crash (try to make it as close to the crash as possible.) This causes all of the bounds-checks described above to be made (it is a pretty slow call). However, if memory has been

corrupted, this call will usually find it. It is another good tool to help find memory corruption problems.

JSEAPIOK

Do not always assume that the ScriptEase ISDK functions succeeded. One problem is to assume a valid result from such a function when it failed, and then using that result, which inevitably causes bad results and crashes. After any ScriptEase API call, you can write the line 'assert(jseApiOK);'. If the call has failed, this assertion will trigger. You can use the 'jseGetLastApiError()' call to get the message associated with the failure.

Common Mistakes

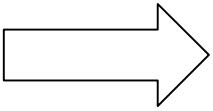
You must always pass the latest jseContext to each library function. The latest context is passed to each call-back function of yours that we call (such as wrapper functions), you need to use that. Don't revert back to some saved context in this situation, as this will cause all kinds of problems. The jseContext is a chain of such contexts, and the core expects to always be working with the last element in this chain. If you pass an older context, presumably now in the middle of the chain, you can expect to crash the ScriptEase core. If you must call an API function using a saved context, you can call the API function 'jseCurrentContext()' to get the end of this chain, which is what you'll want to actually pass to the API function.

Integrating the debugger with your application

The Nombas debugger lets you step through a script line by line, keeping track of the values of variables as they change. Debugging scripts is much easier with this tool.

There are two different ways the debugger can be used: locally and remotely.

- With local execution, the script and the debugger are running on the same machine.
- With remote execution, the application controlling the script communicates with the debugger through a proxy, which runs it though the debugger. This allows you to debug scripts that exist on a remote server.



Types and Macros of the API

This is a list of types and macros used with the ScriptEase:ISDK.

jseActionFlags

DESCRIPTION A set of flags used in a number of ISDK functions to control their behavior in the way variables are searched for and returned from the function.

COMMENTS This type is used in such functions as jseMemberEx, jseGetMemberEx, and jseFindVariable. It controls how the variable is to be returned, and in the case of the member functions, additional options for searching and creating. It is a set of some or all of the following values OR'ed together:

jseCreateVar - If this flag is set, then the variable returned must be explicitly destroyed with jseDestroyVariable(). If this flag is not specified then the variable is tracked internally, and any variable returned from these functions is added to a list of variables to be destroyed when the current context is finished. This can cause problems with long-running persistent contexts because many temporary variables can be added without ever being deleted.

jseDontCreateMember - This only applies to the member functions. If it is set, *NULL* is returned instead of creating the member if it does not exist. Thus, jseGetMember(jsecontext,var,name) is the same as, jseMemberEx(jsecontext,var,name,type,jseDontCreateMember)

jseDontSearchPrototype - This flag only applies to member functions. By default any prototype of the object is searched for members if it is not found in the object itself. If this flag is set, no prototype searching is performed.

jseLockRead - This flag allows finer control over what the returned variable looks like. By default, a reference is returned, and any time a read or write occurs on that variable it must be de-referenced, which could mean calling a dynamic property. If this flag is set the variable is retrieved once when the function is called, but it should only be used for reading from then on. If the variable is found in a _prototype then a new variable is created for writing as an immediate member of the object. If

this is a dynamic object then a new variable is created and the _put function will be called when this variable is destroyed (destruction is explicit with jseCreateVar / jseDestroyVariable) on implicit when this wrapper function returns. This flag and jseLockWrite are mutually exclusive.

jseLockWrite - Similar to jseLockRead, but the variable is locked for writing instead of reading.

SEE ALSO jseFindVariable, jseGetIndexMemberEx, jseGetMemberEx, jseIndexMemberEx, jseMemberEx

jseApiOK

DESCRIPTION A macro used to check that API functions have completed successfully.

COMMENTS This macro is used to check if an error occurred in API calls, i.e., incorrect jseVariables and jseContexts were passed. Its most common use is during initial program development when it follows after an ISDK call (or after a group of calls):

assert(jseApiOK);

SEE ALSO jseGetLastApiError

jseAppLinkFunc

DESCRIPTION This is the type of an user-supplied function provided in the jseExternalLinkParameters structure. It is called by the ScriptEase engine when jseAppExternalLinkRequest() is used to initialize a new context.

SYNTAX

```
typedef jseContext (*jseAppLinkFunc)(
  jseContext jsecontext,
  jsebool initialize);
```

PARAMETERS **jseContext** - The current executing context

initialize - A boolean value passed by the user through the **jseAppExternalLinkRequest()** function signifying whether to initialize a new context or terminate the current one.

COMMENTS This rarely used function is needed by library functions which require a new context to be created from an existing context. This should happen if a library were to create a new context (probably in a new thread) for the application to initialize. The initialize parameter is passed by the user. When creating a context, this value is *True*, indicating that a new context should be created. Otherwise this should be *False*, and the context should be terminated, usually with jseTerminateExternalLink. When initialize is *True*, if *NULL* is returned, it signifies to the engine that there was some sort of error.

RETURNS *NULL* on failure, otherwise a newly created jseContext

SEE ALSO jseAppExternalLinkRequest, jseExternalLinkParameters, jseInitializeExternalLink, jseTerminateExternalLink

jseAtErrorFunc

SYNTAX

```
jseAtErrorFunc (jseContext jsecontext,
                struct AtErrorStruct *info)
```

RETURN void

DESCRIPTION jseAtErrorFunc is immediately called when an error is generated. A structure is passed to it that has information about the error.

```
struct AtErrorStruct
{
   jseVariable errorVariable;
   jsebool trapped;
};
```

The first member is 'errorVariable'. This is the error being generated. It is possible for this not to be an object, but it usually is. For instance, 'throw "foo";' will have the error variable be the string "foo". All error objects can be transformed into strings using 'jseCreateConvertedVariable(...,jseToString)'.

The second member, ‘trapped’, determines whether or not the error will be trapped.

Simply, if it is not trapped, your PrintError function (see jsePrintErrorFunc) will be called immediately after the AtError function. It means that no 'try/catch' handlers are in effect that could possibly catch the error.

SEE ALSO jsePrintErrorFunc; “Error Handling” in the chapter, Integrating the ISDK/C.

jseAtExitFunc

DESCRIPTION The type of an user-defined function passed to jseCallAtExit() which is called when a context is being cleaned up.

SYNTAX

```
typedef void (*jseAtExitFunc)(

   jseContext jsecontext,

   void *Param)
```

PARAMETERS **ntext** - The current executing context

m - The parameter passed to the jseCallAtExit() function.

COMMENTS This is the ScriptEase equivalent to the atexit() C function.

SEE ALSO jseCallAtExit

jseContext

DESCRIPTION Type which represents a calling context for parameters, returns, re-entrance and multitasking.

SYNTAX

```
typedef void * jseContext
```

COMMENTS This is the type of the first parameter for almost every ScriptEase:Integration SDK function.

jseConversionTarget

DESCRIPTION A value that represents a target for converting a variable to another type.

COMMENTS This type is used in the function jseCreateConvertedVariable, where a method of conversion is required. It can be one of the following values:

jseToBoolean - Convert to a boolean value. The contents of the variable depends on the original variable.

- **jseTypeUndefined**
- **jseTypeNull**
- **jseTypeBoolean**
- **jseTypeBuffer**
- **jseTypeNumber**
- **jseTypeString**
- **jseTypeObject**
- **jseToBuffer** - Convert to a buffer type. This conversion is done in the same manner as jseToString, but it is converted to an ASCII sequence of bytes, rather than a Unicode string.

jseToBytes - Convert to a buffer type, but instead of converting each Unicode value to a corresponding ASCII value, a raw transfer of data is done. That is, the Unicode string "Hi" would be converted to the buffer '\0H\0i' or 'H\0i\0', depending on the endianness of the system, and a floating point value would give the actual bytes that share it rather than a test representation of the value.

jseToInt32 - Convert to a 32-bit integer. This is done by converting like jseToInteger does except the range of valid values is 0 to 0xffffffff.

jseToInteger - Convert to an integral type. The value is first converted with jseToNumber. If the result is NaN, then return +0. If the result is +0, -0, +inf, or -inf, return 0. Otherwise, return sign(result) * floor(abs(result)). In other words, the value -4.8 would be converted to -4, shortened to fit. Only values in the range -0x80000000 to 0x7fffffff can be stored.

jseToNumber: Convert to a jseTypeNumber variable based on its type as follows:

- **jseTypeUndefine**: NaN
- **jseTypeNULL**: +0
- **jseTypeBoolean**: The result is 1 if the argument is True. The result is+0 if the argument is False.
- **jseTypeBuffer**: Same as jseTypeString
- **jseTypeNumber**: Same as original

- **jseTypeString**: The string is interpreted as a number, using a complicated set of rules, which are intended to convert human-readable number strings such as "45" and "-45.34" to numbers. If there is an error converting the string to a number, then the result is NaN. More information on these rules can be found in the ECMAScript Language Specifications in section 9.3.
- **jseTypeObject** - Convert input using jseToPrimitive, then convert result with jseToNumber, and return the result.

jseToObject - Convert to an Object. If the original type is jseType*NULL* or jseTypeUndefined, then a runtime error is generated. If the original type is an object, then no conversion is done. Otherwise, the value is converted to the corresponding object wrapper type (i.e. for jseTypeString, new String() will be called with the value as the parameter).

jseToPrimitive - If the variable is any type but jseTypeObject, then no conversion is done. Otherwise, the internal **defaultValue()** function of the object is called and that value returned.

jseToString - Convert to a string based on the following table:

jseTypeUndefined - "undefined"

jseTypeNull - "*NULL*"

jseTypeBoolean - If the argument is *True*, then the result is "True", if the argument is *False*, then the result is "False".

jseTypeString - No conversion done

jseTypeObject - Convert with jseToPrimitive on the object then convert the result with jseToString

jseToUint16 - Convert to an unsigned 16-bit integer. Convert with **jseToInteger**, and then preserve the least significant 16 bits as an unsigned value.

jseToUint32 - Convert to an unsigned 32-bit integer. Convert with **jseToInt32**, and then convert to be unsigned.

SEE ALSO jseCreateConvertedVariable, jseDataType

jseDataType

DESCRIPTION Data type which represents the script type of a jseVariable.

COMMENTS This type is used when getting the type of a variable or creating a new variable of the specified type. This type can be one of these predefined values:

jseTypeBoolean - A boolean value, representing either *True* or *False*

jseTypeBuffer - A buffer, which is an array of bytes. This differs from **jseTypeString** in that strings can be Unicode if Unicode is

enabled.

jseTypeNULL - A *NULL* value, similar to the C constant *NULL*.

jseTypeNumber - A number type, which can be either an integer or a floating point value

jseTypeObject - An object, which can have any number of members (properties).

jseTypeString - A string value.

jseTypeUndefined - An undefined value. This can occur in a number of places, but most of the time it is when a variable has not been declared, or if it has been explicitly set to (void 0).

SEE ALSO jseConvert, jseCreateVariable, jseGetType

jseErrorMessageFunc

DESCRIPTION The type of a function passed in the jseExternalLinkParameters structure which is responsible for printing interpreter errors.

SYNTAX

```
typedef void (*jseErrorMessageFunc) (
   jseContext jsecontext,
   const char *ErrorString);
```

PARAMETERS **jseContext** - The current executing context

ErrorString - A string which is the error to print

COMMENTS This function is called whenever there is an error in the interpreter, either parsing or executing a script.

SEE ALSO jseExternalLinkParameters, jseInitializeExternalLink

jseExternalLibFunc

DESCRIPTION Prototype a ScriptEase wrapper function.

SYNTAX

```
void
jseExternalLibFunc( string functionName );
```

COMMENTS This macro expands to:

```
void  FAR_CALL
functionName(jseContext jsecontext);
```

Use it to write a wrapper function such as:

```
jseExternalLibFunc(myfunc)
{
   /* wrapper function */
}
```

RETURN None.

SEE ALSO jseFuncVarCount, jseGetFunction, jseFuncVarNeed

jseExternalLinkParameters

DESCRIPTION The type of a structure passed to jseInitializeExternalLink() which provides a way to set options and pass needed functions to the interpreter.

SYNTAX

```
struct jseExternalLinkParameters {
   jseErrorMessageFunc  PrintErrorFunc;
   jseFileFindFunc  FileFindFunc;
   jseGetSourceFunc  GetSourceFunc;
   jseMayIContinue  MayIContinue;
   jseAppLinkFunc   AppLinkFunc;
   const char *    jseSecureCode;
   jseLinkOptions  options;
   ulong  hashTableSize;
};
```

COMMENTS This structure is a core part of the interpreter. It must be created and passed to the interpreter through jseInitializeExternalLink() in order for the interpreter to function properly. It has the following fields.

PrintErrorFunc - This function gets called whenever there is an error with the interpreter, either parsing or executing a script. If it is not supplied, then no action is taken when an error occurs, besides the execution being stopped, no error message will be printed. A default version of this function is provided by Nombas, called ToolkitAppPrintError, and it is found in srcapp\printerr.c.

FileFindFunc - This optional function is called by the interpreter when it needs to find a file for inclusion (source files as well as #links and #include files). This acts as a $PATH variable for the interpreter. A default version of this function is provided by Nombas called TookitAppFileSearch and is found in the file srcapp\fsearch.c.

GetSourceFunc - This function is required by the interpreter and is responsible for opening files found by the FileFindFunc, reading lines, and closing files. A default version of this function is provided by Nombas called ToolkitAppGetSourceFunc and is found in the file srcapp\getsource.c.

MayIContinue - This is an optional function and is only used by jseInterpret(). When using the separate, iterative interpreter functions (jseInterpInit(), jseInterpExec(), jseInterpTerm()),

this function is not called. jseInterpret() calls this function before each statement is executed. If this function returns "False", execution is stopped.

AppLinkFunc - This optional function is used when jseAppExternalLinkRequest() is called. It is responsible for initializing the new context (adding any libraries, etc). This allows a toolkit application to obtain a new, initialized context.

jseSecureCode - This null-terminated string is either a filename pointing to a security file or a string containing ScriptEase code to be loaded as a security filter.

options - A mask of options for the interpreter. See jseLinkOptions for more information.

hashTableSize - The size of the internal string hash table used for variable and property names. If this is set to 0, a reasonable default value is used (256). If you are planning to use a great number of strings (for variable or member names), then this small number can cause performance problems and a larger value should be used.

SEE ALSO jseAppLinkFunc, jseErrorMessageFunc, jseFileFindFunc, jseGetExternalLinkParameters, jseGetSourceFunc, jseInitializeExternalLink, jseLinkOptions, jseMayIContinueFunc, jseTerminateExternalLink

jseFindFileFunc

DESCRIPTION This is a user-supplied function for use in the jseExternalLinkParameters structure which is called when the interpreter needs to find an #include or #link file, or when jseInterpret() is called with a filename.

SYNTAX

```
typedef jsebool (*jseFindFileFunc)
```

(jseContext jsecontext, const char *file, char *filePathResults, uint filePathLen, jsebool findLink);

PARAMETERS **jseContext** - The current executing context

fileSpec - A *NULL*-terminated string representing a partial representation of the file. This will be whatever was passed to jseInterpret() or what was in the #include or #link spec.

FilePathResults - A buffer where the function can store the results of the filefind, if a file was found. If this function returns *True* then this will be the value passed to the jseGetSource() function.

filePathLen - Size of the filePathResults buffer.

findLink - *True* if this was included with a #link directive, *False*

otherwise

COMMENTS This function is used to allow the application to adjust the file-specification for a source file. Remember that this doesn't have to be a real file, but can be whatever the application chooses to treat in a file-like way. This function is called before the interpreter begins reading from the source file passed to jseInterpret(), before reading a file described by the #include directive, and before linking with a file defined by the #link directive. Typically, this function will look through paths and other stored values to translate a short file specification into a full file name, similar to the $PATH environment variable.

RETURN *True* if the file was found, *False* otherwise

SEE ALSO jseExternalLinkParameters, jseGetSoruceFunc, jseInitializeExternalLink

jseFuncAttributes

DESCRIPTION A set of flags describing the attributes of a library function.

COMMENTS This OR'ed set of flags is the FuncAttributes member of the jseFunctionDescription structure which is passed to jseAddLibrary(). The following flags are defined:

jseFunc_Default - Default behavior (no flags set)

jseFunc_PassByReference- All parameters passed to this function are passed by reference. By default, variables are passed by value, which means that a copy is made of each parameter before it is passed for the function and any changes to it do not affect the original. If this flag is set, any changes made to parameters affect the original variables as well. This is equivalent to having an '&' (signifying pass-by-reference) before each parameter passed to the function.

jseFunc_Secure - If this flag is set, then it is always safe to call this function. If this flag is not set and Security is enabled, then the function must pass through the security filter.

SEE ALSO jseCreateWrapperFunction, jseFunctionDescription, jseMemberWrapperFunction, jseVarAtributes

jseFunctionDescription

DESCRIPTION A structure defining the way a library function acts. It is passed to jseAddLibrary()

SYNTAX

```
struct jseFunctionDescription {
   const char *FunctionName;
   jseLibraryFunction FuncPtr;
   sword8 MinVariableCount
   sword8 MaxVariableCount;
   jseVarAttributes VarAttributes;
   jseFuncAttributes FuncAttributes;
};
```

COMMENTS This structure must be used any time that a library function is to be added to a jseContext. This structure should never be accessed directly. There are a series of macros which expand into corresponding jseFunctionDescription structures and provide a lot more flexibility, as there are many instances of the jseFunctionDescription structure. Many of these macros share common parameters described here:

FunctionName - The name of the function as a string.

FuncPtr - A pointer to the wrapper function which will get called when the user calls this function.

MinVariableCount - The minimum number of parameters that need to be passed to this function.

MaxVariableCount - The maximum number of parameters that need to be passed to this function. If this value is -1, there is no limit on the number of parameters that can be passed. Also, if jseOptIgnoreExtraParameters is set in the link options, then this parameter is ignored as well, and any extra parameters are ignored (not flagged as errors).

VarAttributes - A set of attributes for the function object (or other variable when using the supplied macros).

FuncAttributes - A set of attributes of the function object.

This structure should never be accessed directly. Instead, the following macros are defined (See the corresponding macro for a description):

JSE_ATTRIBUTE() - Sets the attributes of a named variable

JSE_FUNC_END - Value that MUST be supplied at the end of any table of jseFunctionDescription structures.

JSE_LIBOBJECT() - A library object and object. All subsequent functions defined will be added to this object.

JSE_LIBMETHOD() - Creates a function of the specified name that is a member of the current library object

JSE_PROTOMETH() - Creates a function .prototype.name in the current library object.

JSE_VARASSIGN() - Assign to a member of the current library object from an existing variable.

JSE_VARNUMBER() - Create a numeric member of the current library object with the specified name and value.

JSE_VARSTRING() - Create a member of the current library object with the specified name and value, resulting from evaluating the string.

EXAMPLE

```
jsefloat pi = 3.1415;
static CONST_DATA(struct jseFunctionDescription)
 funcTable[] =
{   JSE_LIBOBJECT("foo",foo_construct,0,0,
       jseDefaultAttr, jseDefaultAttr),
   /* This creates a foo object, with the
    * specified function when it is
    * invoked as new foo()
    */
   JSE_LIBMETHOD("goo",goo_function,1,2,
       jseDontEnum|jseReadOnly,
       jseFunc_PassByReference),
   /* Create a function foo.goo() which
    * takes 1 or 2 parameters, is not
    * enumerated in the foo object, is
    * read only, and receives its
    * arguments by reference
    */
   JSE_PROTOMETH("boo",boo_function,1,-1,
      jseDontDelete,
      jseFunc_Secure),
   /* Create function foo.prototype.boo()
    * which can receive any number of
    * arguments greater than 0, cannot be
    * deleted from foo, and is secure
    */
   JSE_VARNUMBER("pi", &pi,jseReadOnly),
   /* Create read-only number 'foo-pi' with
    * value of 3.1415
    */
   JSE_VARSTRING("car","\"Ford\"",
      jseDefaultAttr),
   /* Create variable 'car' with string
    * value "Ford"
    */
   JSE_FUNC_END
   /* Special value to end table */
};
```

SEE ALSO jseAddLibrary, jseFuncAttributes, jseLibraryFunction, jseVarAttributes, JSE_ATRIBUTE, JSE_FUNC_END, JSE_LIBMETHOD, JSE_LIBOBJECT, JSE_PROTOMETH, JSE_VARASSIGN, JSE_VARNUMBER, JSE_VARSTRING

jseGetSourceFunc

DESCRIPTION This is the type of user-supplied function for use in the jseExternalLinkParameters structure which is called when a file needs to be opened, closed, or read from. Nombas provides a default version of this function called jseToolkitAppSource.

SYNTAX

```
typedef jsebool (*jseGetSourceFunc)(
  jseContext jsecontext, struct jseToolkitAppSource
*ToolkitAppSource, jseToolkitAppSourceFlags flag);
```

PARAMETERS

jseContext - The current executing context

ToolkitAppSource - A structure maintained by the user through calls to this function to keep track of the state of open files.

flag - A flag describing the action to be taken by the function.

COMMENTS This function is used to perform all file I/O and is called by the interpreter in one of three circumstances. The action to be taken is specified by the flags parameter.

jseNewOpen - A new file needs to be opened for reading. The 'name' field of the ToolkitAppSource structure is the name of the file (possibly in shortened form). When this function is called, it must also allocate the 'code' field of the jseToolkitAppSource structure, as the application is responsible for maintaining this pointer. This function call should also verify that the file is readable and store any necessary data in the userdata field of the ToolkitAppSource structure. If this function returns *False*, then no more calls to the function are made with jseGetNext or jseClose, otherwise these calls are guaranteed.

jseGetNext - Read in the next line of file. Set the "code" field to point to the *NULL*-terminated line read. If there are no more lines to read, then return *False*, in which case processing will stop and the function will be called with the flag set to jseClose. Otherwise, processing will continue as normal.

jseClose - This flag is set when a file has been successfully opened and completely read (i.e. call with jseGetNext returned *False*). This is the final call to this callback function, and any cleanup should be done here, including freeing any memory allocated for the 'code' field when jseOpen was called. The return value is ignored.

RETURN *True* if the action was a success, *False* if it failed.

SEE ALSO jseExternalLink, jseInitializeExternalLink, jseToolkitAppSource, jseToolkitAppSourceFlags

jseInterpretMethod

DESCRIPTION A set of flags describing the method for interpreting scripts with jseInterpret().

COMMENTS This data type can be any of the following types OR'ed together:

JSE_INTERPRET_NO_INHERIT - By default, all local variables of the previous context become global variables of the new context. That means that any with statements and the activation object will be propagated and all of these variables will appear as global variables to the program. Setting this flag will turn off this behavior.

JSE_INTERPRET_CALL_MAIN - ECMAScript specifies that any code outside of function definitions is the only code executed, but many C programmers are accustomed to the main() function being called first. If this flag is set, then the main() function is called after any global initialization code.

JSE_INTERPRET_DEFAULT - The default interpret action. No flags are set.

SEE ALSO jseInterpret, jseNewContextSettings

jseLibFunc

DESCRIPTION A macro which expands into a jseLibraryFunction of the specified name.

SYNTAX
```
jseLibFunc(name);
```

PARAMETERS **name** - The name of the library function to create.

COMMENTS This macro expands into the appropriate definition for a jseLibraryFunction with the specified name. For example, jseLibFunc(foo) would expand into (on some systems):
```
void _export _cdecl foo(jseContext jsecontext)
```
and on others to:
```
void foo(jseContext jsecontext)
```

EXAMPLE
```
jseLibFunc(foo_function)
{
   jseVariable foo =
      jseFuncVar(jsecontext,0);
   /* etc etc */
}
```

SEE ALSO jseLibraryFunction

jseLibraryFunction

DESCRIPTION The type of library functions which are added through jseAddLibrary() and subsequently called from scripts.

SYNTAX
```
typedef void (*jseLibraryFunction)(
  jseContext jsecontext);
```

PARAMETERS **jseContext** - The current executing context

SEE ALSO jseLibFunc

jseLibraryInitFunction

DESCRIPTION An optional function passed to jseAddLibrary which is called when initializing the library.

SYNTAX
```
typedef void * (*jseLibraryInitFunction)(
  jseContext jsecontext,
  void _FAR_ *PreviousInstanceLibraryData);
```

PARAMETERS **jseContext** - The current executing context

PreviousIntanceLibraryData - The value the library returned the last time this library was initialized within this context. The first time this is called, it is set to the data passed through jseAddLibrary().

COMMENTS This function is used to initialize any type of library-specific structure that the library may need to access. For example, the library may need to keep track of files opened. In this case they may do something like this:

```
void _FAR_ *
myInitFunc(jseContext jsecontext,
          void _FAR_ *prev)
{
   return (void *) CreateNewFileList();
}
/* Later, in a library function */fileList =
(struct fileList *)
           jseLibraryData(jsecontext);
```

This structure can then be freed in the corresponding jseLibraryTermFunction.

RETURN A void pointer which can later be accessed by any of the library functions using jseLibraryData().

SEE ALSO jseAddLibrary, jseLibraryTermFunction

jseLibraryTermFunction

DESCRIPTION An optional function passed to jseAddLibrary which is called when terminating the library.

SYNTAX

```
typedef void (*jseLibraryTermFunction)(
  jseContext jsecontext,
  void _FAR_ *InstanceData);
```

PARAMETERS **jseContext** - The current executing context

InstanceData - The data returned by the corresponding jseLibraryInitFunction() call.

COMMENTS This function is called when the library is terminated. It can be called multiple times, and every call to the jseLibraryInitFunction will have a matching call to jseLibraryTermFunction. Any initialization performed in the initialization function should be cleaned up here.

SEE ALSO jseAddLibrary, jseLibraryInitFunction

jseLinkOptions

DESCRIPTION A type of a set of flags passed to jseInitializeExternalLink which describe the language options for the interpreter.

COMMENTS **jseOptDefault** - Default behavior (no flags set)

jseOptDefaultCBehavior - If this flag is set, then functions declared without using the 'function' keyword are by default cfunctions.

jseOptDefaultLocalVars - By ECMAScript rules, if a variable is used without an explicit declaration then it is a global variable. This behavior can be changed by setting this flag, in which case all variables declared in such a fashion are local variables unless declared outside of any function.

jseOptIgnoreExtraParameters - If this option is set, any extra parameters passed to a library function beyond the maximum variable count are ignored. By default, this is an error. The rest of the behavior relating to library functions (not passing the minimum number, etc) remains unchanged.

jseOptLenientConversion - This is a flag which specifies a more lenient method of converting variables, getting and putting data in variables, and retrieving function arguments. The first thing that it affects is in using jseFuncVarNeed or jseVarNeed,. If the user uses the JSE_VN_CONVERT() macro, then it is treated as if JSE_VN_ANY were specified in

the from field. In the following example:

```
var = jseFuncVarNeed(jsecontext,
                     0,JSE_VN_CONVERT(
                             JSE_VN_NUMBER,
                             JSE_VN_STRING);
```

If jseOptLenientConversion is not specified and the user passes anything but a number or a string, it is an error. If jseOptLenientConversion is set, then any variable will be converted to a string, in effect changing the JSE_VN_NUMBER to JSE_VN_ANY.

The second way that jseOptLenientConversion affects the ISDK is in the jseGetXXX and jsePutXXX functions. Normally, if the ISDK user calls jsePutLong() on a non-number (or boolean) variable, it is an API error. If this flag is set then the variable is first converted into a number before the data is put in the variable. In the jseGetLong() function, for example, if the source variable is not a number and this flag is not set, it is an API error. If jseOptLenientConversion is set, then a copy of the variable is created, and that copy is converted to a number by calling jseCreateConvertedVariable() with jseToNumber. The numeric value of this copy is returned as the result of the jseGetLong() call. Note that if the ISDK user calls jseGetString(), then a temporary variable is created (to be destroyed when the context is destroyed) in order to ensure that the data remains valid.

jseOptReqFunctionKeyword - If this flag is set, then any function that is declared without the 'function' or 'cfunction' keyword is a runtime error.

jseOptReqVarKeyword - Similar to the jseOptReqFunctionKeyword flag, if this flag is set then any variable that is used without first being declared with the var keyword generates a runtime error.

jseOptToBooleanObjectEval - Without this flag, an object converted to a boolean will always return true, even if that object represents a false value (e.g., new Boolean(false)). With this flag set, objects will be converted to boolean with this logic: ToBoolean(ToPrimitive(object)).

jseOptWarnBadMath - If this flag is set, then any illegal math operations (e.g. dividing by zero) will generate a runtime error. By standard ECMAScript rules, such operations typically return NaN, but no error is generated. This flag should be set if you wish to flag such operations as errors.

SEE ALSO jseExternalLinkParameters, jseGetExternalLinkParameters, jseInitializeExternalLink

jseMayIContinueFunc

DESCRIPTION This is the type of a user-supplied function for use in the jseExternalLinkParameters structure which is called by jseInterpret before processing each statement to determine whether to continue.

SYNTAX

```
typedef jsebool (*jseMayIContinueFunc)(
  jseContext jsecontext);
```

PARAMETERS **jseContext** - The current executing context

COMMENTS This function is provided by the user for use in jseInterpret. If the user is using jseInterpExec/jseInterpTerm, then this function is not necessary as these functions break after each jseInterpExec call. However, when the user is using jseInterpret, this function is called before each statement is processed. If it returns *False*, then execution is immediately stopped and the jseInterpret call exits. Otherwise, execution continues as normal. If the user does not supply this function (i.e. this field is set to *NULL* in the jseExternalLinkParameters structure), then it won't be called and jseInterpret will act as if it always returned success.

SEE ALSO jseExternalLinkParameters, jseInitializeExternalLink, jseInterpExec, jseInterpInit, jseInterpret, jseInterpTerm

jseNewContextSettings

DESCRIPTION A set of flags describing the settings for a new interpret through jseInterpret().

COMMENTS This is an OR'ed mask of the following values:

jseAllNew - Everything created in this context is new and will be destroyed when this context is terminated. All information in the calling context will be preserved. Equivalent of setting all of the other flags.

jseNewAtExit - Create new jseAtExit functions in addition to those currently defined. These new functions will be called when the new context is destroyed. If this flag is not set, any jseAtExit functions will not be called into the calling context is destroyed.

jseNewDefines - All defines within the new context are added in addition to the current defines, but will be removed when the new context is destroyed.

jseNewExtensionLib - Any link libraries used in the new context will be unloaded when the context is destroyed, keeping the original libraries intact.

jseNewFunctions - Any newly declared functions will be removed when the context is destroyed.

jseNewGlobalObject - Create a new (blank) global object for this context.

jseNewLibrary - Create new libraries. This re-initializes all of the libraries added through jseAddLibrary() and calls their respective initialization functions. These libraries are unloaded when the new context is destroyed.

jseNewNone - Nothing is new. Everything that is created or added will remain part of the original context, even after the new context is destroyed. Equivalent to setting no flags.

jseNewSecurity - New security code. Any new security code added will be removed when the new context is destroyed.

SEE ALSO jseInterpret, jseInterpretMethod

jsePrintErrorFunc()

SYNTAX

```
jsePrintErrorFunc (jseContext jsecontext,
                   const char *ErrorString)
```

RETURN void

DESCRIPTION jsePrintErrorFunc is called by the interpreter when a script encounters an untrapped error condition, either parsing or executing a script.

errorString is the message associated with the error.

If this function is set to NULL in your External Parameters structure, no error messages will be printed. This would make running and debugging scripts very difficult, so you almost always should provide a print error function.

Nombas provides a default version of this function, called ToolkitAppFileSearch, which is found in the file srcapp\fsearch.c. Be sure to include the header file fsearch.h for a prototype of this function (This header is automatically included with seall.h).

SEE ALSO jseAtErrorFunc

jseReturnAction

DESCRIPTION The method of returning a jseVariable from a wrapper function.

COMMENTS This can be one of the following values:

jseRetCopyToTempVar - The interpreter will create a new variable, copy to that variable (with jseAssign()), and return the newly created variable. The original variable remains unchanged.

jseKeepLVar - Return a variable that is not to be destroyed when the interpreter is finished with it. This may be because you are returning a variable that was passed as a parameter to your function or the result of a JseMember() call or any other variable that you do not have a lock on (see jseCreateSiblingVariable()).

jseRetTempVar - The variable to be returned will be destroyed by the interpreter when it is no longer needed. This is the most common option, and used when a wrapper function creates a variable to return.

SEE ALSO jseReturnVariable

jseStack

DESCRIPTION Type represents a handle to a jseStack for passing parameters using jseCallFunction.

SEE ALSO jseCreateStack, jseDestroyStack, jsePush, jseCallFunction

jseToolkitAppSource

DESCRIPTION The type of a structure used to maintain the status of open files and manage file I/O through the jseGetSourceFunc supplied in the jseExternalLinkParameters structure.

SYNTAX

```
struct ToolkitAppSource
{
   char * code;
   const char * const name;
   uint   lineNumber;
   void * userdata;
}
```

COMMENTS A structure of this type is passed with every call to jseGetSourceFunc, and file information is shared between that function and the core. In particular, it is used to maintain the state of open files and pass back information read from these files. It has the following fields:

code - The application is responsible for maintaining this field. It should be allocated and freed when the function is called with jseNewOpen and jseClose. When the jseGetSourceFunc is called with jseGetNext, this field should be filled with a '\0'-terminated string containing the next line read from the file.

name - The name of the source file as returned from calling the jseFileFindFunc in the jseExternalLinkParameters structure. The application must not modify this field.

lineNumber - The application can both read to and write from this value. This value is used in error reporting and debugging within the core. This value is initialized to 0 before calling the jseGetSourceFunc with jseNewOpen, and is automatically incremented by one before each call with jseGetNext, so the user need not modify this value if each subsequent call represents the next higher line number.

userdata - This data is reserved for the application and can be used in any manner. Typically it is initialized in jseNewOpen and freed in jseClose and is used to maintain information about the

file (for example, the FILE pointer returned from fopen).

SEE ALSO jseExternalLinkParameters, jseGetSourceFunc, jseInitializeExternalLink

jseToolkitAppSourceFlags

DESCRIPTION The type of one of three values passed to the ToolkitAppSource function in the jseExternalParameters structure passed to jseInitializeExternalLink().

COMMENTS This can be one of the following values. See the description of jseGetSourceFunc for a description of the action to be taken for each value:

jseNewOpen - Open a new file.

jseGetNext - Get next line from a file.

jseClose - Close a file.

SEE ALSO jseGetSourceFunc

jseVarAttributes

DESCRIPTION The type of a mask of attributes for a jseVariable.

COMMENTS This is an OR'ed set of flags describing the attributes of the variable. The flags are as follows:

jseDefaultAttr - The default attributes are used (no flags are set)

jseDontDelete - This variable cannot be deleted. If, within a script, the user calls 'delete [variable]', then no action is taken. This does not affect calls to jseDeleteMember().

jseDontEnum - This variable is not enumerated within for . . . in loops. Therefore, if it is a member of an object and the user enumerates the members of the object using a for . . . in loop, this member will be skipped. jseGetNextMember() always returns all members.

jseImplicitParents - This is an attribute that applies only to local functions. It allows the scope chain to be altered based on the __parent__ property of the 'this' variable. If this flag is set, the __parent__ property is present, and a variable is not found in the local variable context (activation object), then the parents of the 'this' variable are searched (as long as there is a __parent__ property) before searching the global object.

Here is an example, assuming that jseImplicitParents is set on function foo().

```
var a;
```

```
a.value = 4;
var b;
b.__parent__ = a;
b.foo = foo;
b.foo();
function foo()
{
   value = 5;
   // This will actually set a.value to 5
}
```

jseImplicitThis - This attribute applies only to local (script) functions. If this flag is set, then the 'this' variable is inserted into the scope chain before the activation object. This means that if a variable is not found in the local variable context (activation object), the interpreter will then search in the current 'this' variable of the function.

jseReadOnly - This is a read-only variable. Any attempt to write to the variable will fail (nothing will happen).

SEE ALSO jseCreateWrapperFunction, jseFunctionAttributes, jseFunctiondescription, jseGetAttributes, jseMemberWrapperFunction, jseSetAttributes

jseVariable

DESCRIPTION Type of a handle to a jseVariable

jseVarNeeded

DESCRIPTION The type of a mask of flags indicating the type of variable needed, and any conversion that should be done

COMMENTS This value differs from jseDataType in that this is a set of flags, rather then specific values. Be sure to use jseVarNeeded when it is supposed to be used, and not jseDataType. There are also several important features of jseVarNeeded that allow for implicit conversion. The following are acceptable types:

JSE_VN_ANY - Any data type is acceptable.

JSE_VN_BOOLEAN - Only Boolean values are accepted

JSE_VN_BUFFER - Only Buffer type variables are accepted

JSE_VN_BYTE - Only integers between 0 and 255 (one byte) are accepted.

JSE_VN_CONVERT() - A macro which greatly expands the capability of jseVarNeeded and allows for conversion.

JSE_VN_COPYCONVERT - A flag that means to create a copy of the variable and use that rather than the original if the variable must be converted. This is only useful when using the JSE_VN_CONVERT() macro and pass-by-reference, because in this instance you may want a converted variable, but you do not want to change the original to the new type.

JSE_VN_FUNCTION - A case of JSE_VN_OBJECT types in which only function objects are accepted.

JSE_VN_INT - Only numbers which are integer values are accepted (i.e. can be converted to an integer with no loss of precision).

JSE_VN_NOT() - A macro which means 'anything but'. See the JSE_VN_NOT() macro on page for more information.

JSE_VN_NULL - Only *NULL* type variables are accepted

JSE_VN_NUMBER - Only Number variables are accepted

JSE_VN_OBJECT - Only Objects are accepted

JSE_VN_STRING - Only String variables are accepted

JSE_VN_UNDEFINED - Only Undefined variables are accepted.

Here are some examples of usage. See the JSE_VN_CONVERT() macro and JSE_VN_NOT() macro for more information on how those macros work.

```
JSE_VN_NULL
/* Only NULL values */
JSE_VN_NULL|JSE_VN_NUMBER
/* NULL or Number values */
JSE_VN_NOT(JSE_VN_STRING)
/* All types except Strings */
JSE_VN_CONVERT(JSE_VN_BUFFER|
    JSE_VN_STRING,JSE_VN_NUMBER)
/* Accept Strings, buffers, or numbers
 * Strings and buffers are converted
 * to numbers first.
 */
JSE_VN_CONVERT(JSE_VN_STRING,
    JSE_VN_BUFFER)|JSE_VN_COPYCONVERT
/* String or buffer, convert String
 * to buffer and copy it, in case it
 * was passed by reference
 */
```

SEE ALSO jseDataType, jseFuncVarNeed, jseVarNeed, JSE_VN_CONVERT, JSE_VN_NOT

JSE_ATTRIBUTE

DESCRIPTION Macro to create a jseFunctionDescription structure which, when loaded, will set the attributes of the member of the current library object with the specified name.

SYNTAX

```
struct jseFunctiondescription
JSE_ ATTRIBUTE (
   const char *name,
   jseVarAttributes varAttributes)
```

PARAMETERS **name** - The name of the member of the current library object.

varAttributes - See jseFunctionDescription

COMMENTS When loaded, this function description will set the attributes of the member to varAttributes.

EXAMPLE See jseFunctionDescription.

RETURNS jseFunctionDescription structure describing this action.

SEE ALSO jseFunctionDescription

JSECALLFUNCTION

DESCRIPTION jseCallFunction() is now a macro to call jseCallFunctionEx() with JSE_FUNC_DEFAULT as the flags:

SYNTAX

```
#define JSE_FUNC_DEFAULT      0x00
#define JSE_FUNC_TRAP_ERRORS  0x01
```

Exactly analogous to JSE_INTERPRET_TRAP_ERRORS.

```
#define JSE_FUNC_CONSTRUCT    0x02
```

See jseCallFunctionEx() below for details.

```
JSECALLSEQ(jsebool) jseCallFunctionEx(
                              jseContext jsecontext,
                              jseVariable jsefunc,
                              jseStack jsestack,
                              jseVariable *returnVar,
                                 jseVariable thisVar,
                              uint flags);
```

COMMENTS The given 'jsefunc' is a jseTypeObject jseVariable that must be a function. For all possible errors, if JSE_FUNC_TRAP_ERRORS is set, an appropriate Exception object will be returned. If it isn't defined, then 'returnVar' will be NULL and an error message will have been printed (using the normal Error printing scheme.) The jseStack contains the parameters to be passed to the given function. The 'thisVar' is a jseVariable to be given to the function as the 'this' variable. If it is NULL, the global object will be passed. If just calling a function, use NULL. When trying to call a particular Object's member function (for instance, the 'toString' method of an object), you'll want to pass that Object as the 'thisVar'

Note that even with JSE_FUNC_TRAP_ERRORS, the return can be NULL if you called the function illegally (such as 'jsefunc' not being a function.) Use jseGetLastApiError() to find the problem.

The JSE_FUNC_CONSTRUCT flag will allow you to call a constructor. The given 'jsefunc' must actually have a constructor associated with it, else an error will be generated. In this case, the 'thisVar' is ignored, since a call to a constructor generates a new Object for that constructor to fill in.

When calling an Object's member function, use jseGetMember() to get the jseVariable associated with the function you'd like to call. When trying to call a generic function or constructor, it is usually easiest to use jseFindVariable() to look up a function you'd like to call.

As a convenience, the 'jsestack' can be NULL if there are no parameters to be passed to the function.

JSE_ENGINE_VERSION_ID

DESCRIPTION A predefined value representing the current version of the ISDK, used to check the value returned from jseInitializeEngine().

COMMENTS This value represents the current version of the ISDK, and is used to check that the engine version is the same as the expected value. Typically, this will appear as follows:

```
if( JSE_ENGINE_VERSION_ID !=
    jseInitializeEngine() )
   PrintError("Wrong version of ISDK!");
```

SEE ALSO jseInitializeEngine

JSE_FUNC_END

DESCRIPTION Macro which returns a jseFunctionDescription structure signifying the end of a description table.

SYNTAX
```
JSE_FUNC_END
```

EXAMPLE See jseFunctionDescription.

RETURNS jseFunctionDescription structure signifying end.

SEE ALSO jseFunctionDescription

JSE_LIBOBJECT

DESCRIPTION Macro to create a jseFunctionDescription structure describing an object.

SYNTAX
```
struct jseFunctionDescription
JSE_LIBOBJECT(
   const char *name,
   jseLibraryFunction callFunction,
   sword8 MinVariableCount,
   sword8 MaxVariableCount,
   jseVarAttributes varAttributes,
   jseFuncAttributes funcAttributes)
```

PARAMETERS

name - The name of the object to create

callFunction - The library function to call when the object is instantiated by calling name as a function (i.e. new name()).

MinVariableCount - See jseFunctionDescription

MaxVariableCount - See jseFunctionDescription

varAttributes - See jseFunctionDescription

funcAttributes - See jseFunctionDescription

COMMENTS After using JSE_LIBOBJECT() within a function description table, every subsequent call to JSE_LIBMETHOD(), JSE_PROTOMETH(), JSE_VARASSIGN(), JSE_VARSTRING(), and JSE_VARNUMBER() will use this object to add to.

EXAMPLE See jseFunctionDescription.

RETURNS jseFunctionDescription structure describing this object.

SEE ALSO jseFunctionDescription

JSE_LIBMETHOD

DESCRIPTION Macro to create a jseFunctionDescription structure describing a function method

SYNTAX

```
struct jseFunctiondescription
JSE_LIBMETHOD(
   const char *name,
   jseLibraryFunction function,
   sword8 MinVariableCount,
   sword8 MaxVariableCount,
   jseVarAttributes varAttributes,
   jseFuncAttributes funcAttributes)
```

PARAMETERS

name - See jseFunctionDescription

function - See jseFunctionDescription

MinVariableCount - See jseFunctionDescription

MaxVariableCount - See jseFunctionDescription

varAttributes - See jseFunctionDescription

funcAttributes - See jseFunctionDescription

COMMENTS This macro will add the function to the current library object, either the one created with the last call to JSE_LIBOBJECT, or if that has not been used yet in the table, the object supplied with jseAddLibrary().

EXAMPLE See jseFunctionDescription.

RETURNS jseFunctionDescription structure describing this function.

SEE ALSO jseFunctionDescription

JSE_PROTOMETH

DESCRIPTION Macro to create a jseFunctionDescription structure describing a prototype function

SYNTAX

```
struct jseFunctionDescription
JSE_PROTOMETH(
   const char *name,
   jseLibraryFunction function,
   sword8 MinVariableCount,
   sword8 MaxVariableCount,
   jseVarAttributes varAttributes,
   jseFuncAttributes funcAttributes)
```

PARAMETERS

name - See jseFunctionDescription

function - See jseFunctionDescription

MinVariableCount - See jseFunctionDescription

MaxVariableCount - See jseFunctionDescription

varAttributes - See jseFunctionDescription

funcAttributes - See jseFunctionDescription

COMMENTS This macro will add the function to the prototype of the current library object, either the one created with the last call to JSE_LIBOBJECT, or if that has not been used yet in the table, the object supplied with jseAddLibrary(). This is equivalent to '.prototype.name' of the current object.

EXAMPLE See jseFunctionDescription.

RETURNS jseFunctionDescription structure describing this function.

SEE ALSO jseFunctionDescription

JSE_VARASSIGN

DESCRIPTION Macro to create a jseFunctionDescription structure which, when loaded, will assign to a member of the current object from a global variable.

SYNTAX

```
struct jseFunctiondescription
JSE_VARASSIGN(
   const char *memberName,
   const char *globalName,
   jseVarAttributes varAttributes)
```

PARAMETERS **memberName** - The name of the member of the current library object to assign to.

globalName - The name of the global variable to assign from.

varAttributes - See jseFunctionDescription

COMMENTS When loaded, this function description will perform the assign as if the statement "[libobject].memberName = globalName" had been executed within a script.

EXAMPLE See jseFunctionDescription.

RETURNS jseFunctionDescription structure describing this action.

SEE ALSO See jseFunctionDescription

JSE_VARSTRING

DESCRIPTION Macro to create a jseFunctionDescription structure which, when loaded, will create a member of the current library object with the specified string evaluated as its value.

SYNTAX

```
struct jseFunctiondescription
JSE_VARSTRING(
   const char *name,
   const char *evalstring,
   jseVarAttributes varAttributes)
```

PARAMETERS **name** - The name of the member of the current library object to assign to.**evalstring -** The string to be evaluated and assigned to the member.

varAttributes - See jseFunctionDescription

COMMENTS When loaded, this function description will perform the assign as if the statement ' [libobject].name = "string" ' had been executed within a script.

EXAMPLE See jseFunctionDescription.

RETURNS jseFunctionDescription structure describing this action.

SEE ALSO jseFunctionDescription

JSE_VARNUMBER

DESCRIPTION Macro to create a jseFunctionDescription structure which, when loaded, will create a member of the current library object with the specified number as its value.

SYNTAX

```
struct jseFunctionDescription
JSE_ VARNUMBER(
   const char *name,
   jsenumber *number,
   jseVarAttributes varAttributes)
```

PARAMETERS **name** - The name of the member of the current library object to assign to.

number - A pointer to the number value to assign to the member.

varAttributes - See jseFunctionDescription

COMMENTS When loaded, this function description will assign the value pointed to by number to the member.

EXAMPLE See jseFunctionDescription.

RETURNS jseFunctionDescription structure describing this action.

SEE ALSO jseFunctionDescription

JSE_VN_CONVERT

DESCRIPTION Macro to create a jseVarNeeded set of flags describing a conversion to be performed.

SYNTAX

```
jseVarNeeded
JSE_VN_CONVERT(
   jseVarNeeded sourceTypes,
   jseVarNeeded destType)
```

PARAMETERS

sourceTypes - OR'ed set of jseVarNeeded flags representing the acceptable source types for conversion.

destType - A jseVarNeeded flag representing the destination type to convert to.

COMMENTS The JSE_VN_CONVERT() macro expands the capability of the jseVarNeeded flags to perform automatic conversion between types. See the jseVarNeeded type for more information on these flags. If the variable to be retrieved matches one of the flags in sourceTypes, then it is converted to the destType. This conversion is done in-place, converting the original variable, unless JSE_VN_COPYCONVERT is OR'ed with the JSE_VN_CONVERT() macro (see jseVarNeeded description). The conversion happens in the standard ECMAScript manner, calling jseCreateConvertedVariable() with the appropriate jseConversionTarget type. The result replaces the original unless JSE_VN_COPYCONVERT is OR'ed in to the flag set. If the jseOptLenientConversion flag is set in the jseLinkOptions of the current context, then the sourceTypes field is assumed to be JSE_VN_ANY, regardless of what is specified. Also, only basic types are allowed as parameters to JSE_VN_CONVERT(). That means the following jseVarNeeded values are unacceptable: JSE_VN_FUNCTION, JSE_VN_BYTE, JSE_VN_INT, JSE_VN_COPYCONVERT. This means you cannot call JSE_VN_CONVERT(JSE_VN_INT,JSE_VN_STRING). You must use JSE_VN_NUMBER instead.

EXAMPLE See jseVarNeeded.

SEE ALSO jseFuncVarNeed, jseVarNeed, jseVarNeeded, JSE_VN_NOT

JSE_VN_NOT

DESCRIPTION Macro to create a set of jseVarNeeded flags to represent every type except for the ones specified as parameters.

SYNTAX
```
jseVarNeeded
JSE_VN_NOT(jseVarNeeded types)
```

PARAMETERS **types** - An OR'ed set of jseVarNeeded flags to exclude from the resulting jseVarNeeded

COMMENTS This is equivalent to specifying JSE_VN_ANY without the specified types.

EXAMPLE See jseVarNeeded.

SEE ALSO jseFuncVarNeed, jseVarNeed, jseVarNeeded, JSE_VN_CONVERT

API Functions

The following functions call APIs make up the ScriptEase:ISDK:

jseActivationObject

DESCRIPTION Get the local variable object for the function currently being executed.

SYNTAX
```
jseVariable
jseActivationObject(jseContext jsecontext);
```

COMMENTS **jseContext** - The current executing context

RETURN This function returns the current activation object, or local variable object, of the last local (script) function. Thus the local variable "a" of the last script function would be a member of this object.

SEE ALSO jseGlobalObject

jseAddLibrary

DESCRIPTION Add an external function library to a given jseContext.

SYNTAX
```
void
jseAddLibrary(jseContext jsecontext,
              const jsechar * objectVariableName
                const struct jseFunctiondescription
                *functionTable,
              void *InitLibData,
              jseLibraryInitFunction
                libInitFunction,
              jseLibraryTermFunction
                libTermFunction );
```

PARAMETERS **jseContext** - The current executing context

objectVariableName - The name of the object the properties will be associated with. If *NULL* is supplied in this field, then the global object will be used.

functionTable - An array of function descriptions to add to the context.

initLibData - This data is passed the first time that the libInitFunction is called. Most of the time this can be set to *NULL* unless you have special needs. If libInitFunction is *NULL* this will also be the data available to each library function via jseLibraryData().

libInitFunction - This function will be called each time the library is loaded (which could occur more than once if jseInterpret is called with the appropriate settings). The pointer this function returns is available to all wrapper functions via jseLibraryData().

libTermFunction - This function will be called when the library is being unloaded, which means that it is no longer being used. Again, this could happen more than once.

COMMENTS Use this function to add library functions to the jseContext. A static table of jseFunctionDescription structures is defined, and this table is passed as the second parameter to the function. Here is an example of usage:

```
static struct
CONST_DATA(struct jseFunctiondescription)
   myFuncs[ ] =
{
   JSE_LIBMETHOD("foo",fooFunc,0,0, jseDefaultAttr,
                 jseDeafultAttr),
   JSE_FUNC_END
};

/* Add some initialization function */
jseAddLibrary(jsecontext,"myFuncs",
              myFuncs,NULL,NULL,NULL);
.
```

RETURN None.

SEE ALSO jseFunctionDescription, jseLibraryData, jseLibraryInitFunction, jseLibraryTermFunction

jseAppExternalLinkRequest

DESCRIPTION Create a new jseContext using the jseAppLinkFunc provided in the jseExternalLinkParameters structure.

SYNTAX
```
jsecontext
jseAppExternalLinkRequest( jseContext jsecontext,
                           jsebool Initialize)
```

PARAMETERS **jseContext** - The current executing context.

Initialize - The second initialization parameter passed to the AppLink function.

COMMENTS If the user provides a jseAppLinkFunction when initializing the jseExternalLinkParameters structure, this function will just pass the call along and retrieve a new jseContext that is your function's return. This function should first be called with *True* as the second parameter to initialize a new context and then be called a second time with *False* in order to clean up the returned context.

EXAMPLE
```
newcontext = jseAppExternalLinkRequest( jsecontext,
                                        True);
if(newcontext == NULL )
    PrintError( "Initialization failed");
    /* ... Use the new context here ... */
    (jseAppExternalLinkRequest(newcontext, False);
```

RETURNS *NULL* on failure, otherwise a valid jseContext.

NOTE This function is not used frequently; usually if a link library forces an interpret in a new context initialized in an application-defined way.

SEE ALSO jseAppLinkFunc, jseExternalLinkParameters, jseInitializeExternalLink

jseAssign

DESCRIPTION Copy the current value of one variable to another.

SYNTAX
```
jsebool
jseAssign(jseContext jsecontext,
          jseVariable destVar,
          jseVariable srcVar );
```

PARAMETERS **jseContext** - The current executing context

destVar - The ScriptEase variable to set

srcVar - The ScriptEase variable to assign from.

This function assigns the value of the ScriptEase data defined by destVar to be equivalent to the value of the ScriptEase data defined by srcVar. The result of this function is the type of operation performed by the ' = ' operator.

RETURN return boolean *True* for success, else return *False* if the assignment was unsuccessful.

SEE ALSO jseGetType, jseConvert, jseCreateConvertedVariable

jseBreakpointTest

DESCRIPTION Test to see if the current line is a valid breakpoint.

SYNTAX

```
jsebool
jseBreakpointTest(jseContext jsecontext,
                  const char  *FileName,
                  uword32 lineNumber);
```

PARAMETERS **jseContext** - The current executing context

FileName - Name of the file to be tested.

LineNumber - The line number to check for breakpoint

COMMENTS Check if currently-running script thinks it has a breakpoint in this file at this lineNumber. This function is provided to facilitate debugging.

RETURN return *True* if on a valid breakpoint, else return *False*.

SEE ALSO jseLocateSource

jseCallAtExit

DESCRIPTION Add a function to be called when exiting a jseContext

SYNTAX

```
void
jseCallAtExit(jseContext jsecontext,
              jseAtExitFunc exitFunction,
              void *param );
```

PARAMETERS **jseContext** - The current executing context

exitFunction - The function to call at exit.

param - A parameter to give to the atexit() function when it is called.

COMMENTS This function sets a function to be called when a top-level interpreted jseContext is destroyed, similar to the C function atexit(). Any number of functions may be registered with jseCallAtExit(); they will be called in the reverse order in which they're added. At-exit functions are called regardless of the reason for the exit. If an error condition exists, the error flag will be turned off while calling these functions. These functions will be called before any libraries added with jseAddLibrary are terminated.

RETURN None.

SEE ALSO jseAtExitFunc

---

jseCallFunction

DESCRIPTION Call a ScriptEase function.

SYNTAX

```
jsebool
jseCallFunction(jseContext jsecontext,
                jseVariable jsefunction,
                jseStack jsestack,
                jseVariable  *returnVar,
                jseVariable thisVar);
```

PARAMETERS **jseContext** - The jseContext in which to call the specified function.

jsefunction - The jseVariable to the specified ScriptEase function.

jsestack - The parameters to pass to the specified function.

returnVar - A pointer to a jseVariable which will be filled in with the return variable from the function. This variable should not be created beforehand with jseCreateVariable(), as it will be replaced, and will cause a memory leak. Also, this variable will automatically be destroyed when the stack is destroyed (unless explicitly popped off the stack).

thisVar - The jseVariable to be used as the 'this' var; use *NULL* for the global object.

COMMENTS This function is used to make a call to a ScriptEase function from within your application. The function passed in the **jsefunction** parameter is typically a value returned from jseGetFunction.

RETURN returns *True* if the call was successful, *False* otherwise. The context error flag will have been cleared when this function returns. Therefore one should use this return value to determine if the function failed.

SEE ALSO jseCurrentFunctionName, jseGetFunction, jseCreateStack, jseDestroyStack, jsePush

jseClearApiError

DESCRIPTION Clear error message set by failed API call.

SYNTAX
```
void
jseClearApiError()
```

COMMENTS If one of the API calls fails (if an incorrect parameter type was passed, for example), it will set an error message explaining the reason for the failure. This function erases the error message.

SEE ALSO jseApiError, jseGetLastApiError

jseCompare

DESCRIPTION Compare two script variables for greater-than, less-than or equal comparison.

SYNTAX
```
jsebool
jseCompare(jseContext jsecontext
           jseVariable variable1,
           jseVariable variable2,
           slong *compareResult);
```

PARAMETERS **jseContext** - The current executing context

variable1 - The first variable to compare.

variable2 - The second variable to compare.

compareResult - On return, this variable will be set to:
< 0 if variable 1 is less than variable 2
0 if variable 1 is equal to variable 2
> 0 if variable 1 is greater than variable 2

COMMENTS This routine compares two jseVariables. In its most basic form, it simply compares if two variables are equal, in that the data they contain are equivalent, or that they point to the same object. In addition, one of the following predefined values can be passed as compareResult to use the standard ECMAScript comparison routines:

JSE_COMPEQUAL - Compare using ECMAScript equality rules.

JSE_COMPLESS - Compare using ECMAScript less-than rules (different from equality rules).

Typically, to do ECMAScript comparisons, the user should never call this function directly. Use the functions jseCompareLess() and jseCompareEquality(), which map to the equivalent flags above.

RETURN In a standard comparison, indicates whether the comparison was successful. When using JSE_COMPEQUAL, returns a boolean value

as to whether the two variables are equal. When using JSE_COMPLESS, returns a boolean value as to whether the first variable is less than the second variable.

SEE ALSO jseEvaluateBoolean, jseAssign, jseCompareLess, jseCompareEquality

jseCompareEquality

DESCRIPTION Compare two script variables for equality using ECMAScript rules.

SYNTAX
```
jsebool
jseCompareEquality(jseContext jsecontext
                   jseVariable variable1,
                   jseVariable variable2);
```

PARAMETERS **jseContext** - The relevant jseContext.

variable1 - The first variable to compare.

variable2 - The second variable to compare.

COMMENTS This function is equivalent to calling jseCompare with the result value JSE_COMPEQUAL.
If one variable is a string and the other a number, the string will be converted to a number before comparing. Boolean values will be converted to numbers before being compared.

RETURN *True* if the variables are equal to each other, *False* if they are not.

SEE ALSO jseEvaluateBoolean, jseAssign, jseCompare, jseCompareLess

jseCompareLess

DESCRIPTION See if one variable's value is less than another's, using ECMAScript rules.

SYNTAX
```
jsebool
jseCompareLess(jseContext jsecontext
               jseVariable variable1,
               jseVariable variable2);
```

PARAMETERS **jseContext** - Current context

variable1 - The first variable to compare.

variable2 - The second variable to compare.

COMMENTS This function converts variables to primitive values before they are compared. If the two variables are both strings, they will be compared as strings; otherwise they will be converted to numbers and compared.

RETURN *True* if variable1 is less than variable2, *False* if variable1 is greater than or equal to variable2.

SEE ALSO jseEvaluateBoolean, jseAssign, jseCompareEquality, jseCompare

jseConvert

DESCRIPTION Convert a variable to a new jseDataType.

SYNTAX
```
void
jseConvert(jseContext jsecontext,
           jseVariable variable,
           jseDataType dType );
```

PARAMETERS **jseContext** -The current executing context.

variable - The ScriptEase variable to convert.

dType - The data type the variable is being converted to.

COMMENTS This function changes a variable from one type to another. This function does not preserve the current contents of the variable, but instead is much like destroying the previous variable and creating a new variable with this type. If the variable is already of the specified type, no conversion is performed and no data is lost.

RETURN None.

SEE ALSO jseDataType, jseGetType, jseAssign

jseCopyBuffer

DESCRIPTION Copy a section of a buffer from a jseVariable to a local buffer.

SYNTAX
```
ulong
jseCopyBuffer(jseContext jsecontext,
              jseVariable variable,
              void *buffer,
              ulong start,
              ulong length);
```

PARAMETERS **jseContext** - The current executing context.

variable - The buffer variable which contains the data to be copied.

buffer - The local buffer that will be filled with the copied data.

start - The offset within the variable where the copying will start from.

length - The length of data to be copied from the buffer variable.

RETURN None.

SEE ALSO jseCopyString, jseGetBuffer

jseCopyString

DESCRIPTION Copy string data from a jseVariable to a user allocated buffer.

SYNTAX

```
ulong
jseCopyString(jseContext jsecontext,
              jseVariable variable,
              jsechar *buffer,
              ulong start,
              ulong length);
```

PARAMETERS **jseContext** - The current executing context.

variable - The variable containing the string to be copied.

buffer - The buffer which will be filled with the string data

start - The offset in the variable of the first character to be copied.

length - The length of the string to be copied from the variable.

RETURN None.

SEE ALSO jseCopyBuffer, jseGetString

jseCreateCodeTokenBuffer

DESCRIPTION Compile a block of ScriptEase code into executable tokens

SYNTAX

```
void *
jseCreateCodeTokenBuffer(jseContext jsecontext,
                         const char  *source,
                         jsebool sourceIsFileName,
                         uint  *bufferLen);
```

PARAMETERS **jseContext** - The current executing context.

source - ScriptEase source code to tokenize.

sourceIsFileName - *True* if Source is a filename, else *False* if Source is a block of code.

bufferLen - This argument is set by the call to the length of the created buffer.

COMMENTS This call will compile the code in the source parameter into a binary sequence of tokens which can later be executed with jseInterpret or jseInterpInit by passing the returned buffer as the tokenized code parameter

RETURN The return value is a void pointer to the block of tokens.

SEE ALSO jseInterpret, jseInterpInit

jseCreateConvertedVariable

DESCRIPTION Create a new variable from another variable and convert its data.

SYNTAX
```
jseVariable
jseCreateConvertedVariable(jseContext jsecontext,
                  jseVariable variableToConvert
                     jseConversionTarget targetType);
```

PARAMETERS **jseContext** - The current executing context.

variableToConvert - variable to be used as a model for the new variable.

targetType - type of variable to convert to.

COMMENTS This function will convert the **variableToConvert** into a variable of the new targetType using the standard ECMAScript conversion rules. See the description of jseConversionTarget for a description of these rules. This differs from jseConvert in that it uses ECMAScript conversion, rather than simply erasing the data and creating a blank type.

RETURN If successful, a pointer to the converted jseVariable created. If there is not enough system memory to create the variable (extremely unlikely), *NULL* will be returned. You must destroy the variable using jseDestroyVariable when you are done with it.

SEE ALSO jseConvert, jseCreateVariable, jseCreateSiblingVariable, jseDestroyVariable, jseConversionTarget

jseCreateFunctionTextVariable

DESCRIPTION Return the source text of a function.

SYNTAX
```
jseVariable
jseCreateFunctionTextVariable(jseContext jsecontext,
                        jseVariable functionVariable);
```

PARAMETERS **jseContext** - Current jseContext

functionVariable - Variable to get the source from.

COMMENTS This function takes a variable and returns the source text of the function. This is equivalent to calling ToString() on the function. You must destroy the variable using jseDestroyVariable when you are done with it.

RETURN Returns a string containing the source text of **functionVariable**.

SEE ALSO jseCreateVariable, jseCreateSiblingVariable, jseDestroyVariable

jseCreateLongVariable

DESCRIPTION Shortcut to create a ScriptEase variable of an integer value.

SYNTAX
```
jseVariable
jseCreateLongVariable(jseContext jsecontext,
                      slong value );
```

PARAMETERS **jseContext** - The current executing context.

value - Value to initialize this ScriptEase variable to.

COMMENTS This function creates a ScriptEase variable of type jseTypeNumber and puts the specified value in the variable. This is equivalent to creating a variable of type jseTypeNumber and then calling jsePutLong() to put a value into it.

RETURN If successful, a pointer to the jseVariable created. If there is not enough system memory to create the variable (extremely unlikely), *NULL* will be returned.

SEE ALSO jseCreateVariable, jseCreateSiblingVariable, jseCreateConvertedVariable, jseDestroyVariable, jsePutLong

jseCreateSiblingVariable

DESCRIPTION Create a ScriptEase Sibling Variable.

SYNTAX
```
jseVariable
jseCreateSiblingVariable(jseContext jsecontext,
                jseVariable olderSiblingVar,
                slong elementOffsetFromOlderSibling);
```

PARAMETERS **jseContext** - The current executing context.

olderSiblingVar - The variable that you are basing the new sibling variable on.

elementOffsetFromOlderSibling - The index into the array you are creating this sibling variable from (if a string or buffer).

COMMENTS This routine creates a sibling ScriptEase Variable. A sibling variable is a variable that references an already existing ScriptEase Variable. Changes to sibling variables affect each other. The offset parameter is used in conjunction with buffer and string variables, as it specifies an offset into the data at which to begin the sibling variable. The original variable and the sibling variable still reference the same variable, but calling jseGetString on the new variable will start at the new offset into the original.

EXAMPLE

```
jseVariable original = jseCreateVariable(
                                jsecontext, jseTypeString);
jsePutString(jsecontext,original,"one two");
jseVariable new = jseCreateSiblingVariable(
                                jsecontext,original,4);
jsechar * data = jseGetString(jsecontext, new);
```

Data now points to "two", and any changes to original or new will affect the other.

RETURN If successful, a pointer to the sibling jseVariable created. If there is not enough system memory to create the variable (extremely unlikely), *NULL* will be returned.

SEE ALSO jseCreateVariable, jseCreateConvertedVariable, jseCreateLongVariable, jseDestroyVariable

jseCreateStack

DESCRIPTION Create a jseStack.

SYNTAX

```
void  *
jseCreateStack(jseContext jsecontext);
```

PARAMETERS **jseContext** - The current executing context.

COMMENTS This function creates a jseStack which is used for pushing parameters and calling functions from within the ISDK.

RETURN Returns a pointer to the new jseStack. *NULL* will be returned if there is insufficient memory to create the stack.

SEE ALSO jseCallFunction, jseDestroyStack, jsePush

jseCreateVariable

DESCRIPTION Create a jseVariable of a given type.

SYNTAX

```
jseVariable
jseCreateVariable(jseContext jsecontext,
                  jseDataType VType);
```

PARAMETERS **jseContext** - The current executing context.

VType - The type of ScriptEase variable to create.

RETURN If successful, a pointer to the jseVariable created. If there is not enough system memory to create the variable (extremely unlikely), *NULL* will be returned.

SEE ALSO jseCreateSiblingVariable, jseCreateConvertedVariable, jseCreateLongVariable, jseDestroyVariable

jseCreateWrapperFunction

DESCRIPTION Create a variable object that is a callable function.

SYNTAX

```
jseVariable
jseCreateWrapperFunction(jseContext jsecontext,
                         const char *functionName
                         jseLibraryFunction funcPtr,
                         sword8 minVariableCount,
                         sword8 maxVariableCount,
                         jseVarAttributes varAttributes,
                         jseVarAttributes funcAttributes,
                         void * fData);
```

PARAMETERS **jseContext** - The current executing context

functionName - The name of the function to be created.

funcPtr - A pointer to the library function which will be called when this function is called.

minVariableCount - The minimum number of variables that can be passed to the function.

maxVariableCount - The maximum number of variables that can be passed to the function.

varAttributes - attributes of the new variable. See jseVarAttributes, Types & Macros chapter, for more information.

FuncAttributes - attribute of function.

fData - pointer that will be available to the wrapper function via jseLibraryData().

RETURN If successful, this returns the jseVariable created. If there is not enough system memory to create the variable (extremely unlikely), *NULL* will be returned. This variable must be destroyed by calling jseDestroyVariable() when you are done with it. The variable is a function object which will call your wrapper function.

jseCurrentContext

DESCRIPTION Return the current jseContext based on any level of previous context.

SYNTAX

```
jseContext
jseCurrentContext(jseContext ancestorContext);
```

COMMENTS Pass this function an old jseContext and it will return the most current descendent of it.

ancestorContext - Any previous level jseContext.

The current context for the current thread of execution.

RETURN

SEE ALSO jsePreviousContext

jseCurrentFunctionName

DESCRIPTION Get the currently executing ScriptEase function.

SYNTAX

```
const jsechar  *
jseCurrentFunctionName(jseContext jsecontext);
```

COMMENTS Returns a pointer to the name of the function currently executing. Do not alter the Returned string.

jseContext - The context to use for this interpret.

RETURN Pointer to the name of the function currently executing. DO NOT modify this string.

SEE ALSO jseGetFunction, jseCurrentFunctionVariable

jseCurrentFunctionVariable

DESCRIPTION Get the current variable associated with a function

SYNTAX

```
jseVariable
jseCurrentFunctionVariable(jseContext jsecontext);
```

COMMENTS Returns the function object of the function currently executing. If it is the initialization function , then `NULL` is returned because there is no function object.

Function object of the function currently executing.

RETURN

jseCurrentFunctionName

SEE ALSO

jseDeleteMember

DESCRIPTION Delete a jseObject property.

SYNTAX

```
void
jseDeleteMember(jseContext jsecontext,
                jseVariable objectVar,
                const jsechar  * name );
```

COMMENTS This function deletes a property of an object. This function ignore the jseDontDelete attribute (which is only used for the 'delete' operator within scripts).

jseContext - The current executing context.

objectVar - ScriptEase variable pointer.

name - The name of the object property to delete.

RETURN None.

SEE ALSO jseGetMember, jseGetNextMember

jseDestroyStack

DESCRIPTION Destroy a jseStack.

SYNTAX

```
void
jseDestroyStack(jseContext jsecontext,
                jseStack stack);
```

COMMENTS This function destroys the specified stack.

jseContext - The current executing context.

stack - The stack to destroy

RETURN None.

SEE ALSO jseCallFunction, jseCreateStack, jsePush

jseDestroyVariable

DESCRIPTION Destroy a ScriptEase variable.

SYNTAX

```
void
jseDestroyVariable(jseContext jsecontext,
                   jseVariable variable );
```

COMMENTS Use this routine to free up the system resources allocated to a ScriptEase variable when it is no longer needed. Variables created with one of the jseCreateXXX() functions must be destroyed; variables created with the jsereturnXXX() functions need only be destroyed if the third parameter passed to the function is not jseReturnTempVariable.

This is probably the most confusing concept in the API. Destroying a variable does not mean destroying the contents of the variable. Instead, it means destroy your handle or lock on the variable. If this is the last such lock, then the contents are destroyed.
When you get a jseVariable handle, sometimes it is a lock that you must destroy, sometimes you must not destroy it. The description of the API function will specify which case it is, but the general rule is that if the API function has the word 'create' in it, you are getting a lock you must destroy.
The API jseReturnVar() in several modes accepts a lock that it will destroy when it is done; by passing the variable to it, you are transferring your lock. If you have a variable that you aren't supposed to destroy and pass it to this function, you will have a problem. Either use jseRetKeepLVar to tell jseReturnVar() not to destroy the variable or create a lock using jseCreateSiblingVariable() which you can then pass to it.

jseContext - The current executing context.

variable - The ScriptEase variable to destroy.

RETURN None.

SEE ALSO jseCreateVariable, jseCreateSiblingVariable, jseCreateConvertedVariable, jseCreateLongVariable, jseReturnVar

jseEvaluateBoolean

DESCRIPTION Determine if a ScriptEase Variable is *True* or *False*.

SYNTAX
```
jsebool
jseEvaluateBoolean(jseContext jsecontext,
                   jseVariable variable);
```

COMMENTS Test to see if a ScriptEase variable evaluates to *True* or *False*. Pass a variable of type jseTypeBoolean.

jseContext - the context that the tested variable belongs to.

variable - The ScriptEase variable to test.

RETURN The boolean value of variable.

jseFindVariable

DESCRIPTION Search for a variable with a given name.

SYNTAX
```
jseVariable
jseFindVariable(jseContext jsecontext,
                const char * name);
```

COMMENTS **jseContext** - the context that the tested variable belongs to.

variable - The name of the variable sought.

This variable searches the current scope chain for a variable with the given name. Usually, you want to search the scope chain as it was for the function that called you, since someone will likely write something like:

```
function myfunc()
{
  var a;
  wrapper("a");
}
```

The 'a' refers to the 'a' from the point of view of the calling function, not your wrapper function (which does not have the locals of the calling function as part of its scope chain.) In most cases, thus, the correct way to call this function is to use 'jsePreviousContext(jsecontext)' as the context you pass to this function.

RETURN Returns the variable if it is found, *NULL* if no such variable can be found.

SEE ALSO jseGetVariableName

jseFuncVar

DESCRIPTION Get a ScriptEase function wrapper argument.

SYNTAX
```
jseVariable
jseFuncVar(jseContext jsecontext,
           uint ParameterOffset );
```

COMMENTS This function gets a parameter passed to a wrapper function. It returns a jseVariable pointer, but does no type checking.

jseContext - The context for this wrapper function. Use the value supplied to the wrapper function by the ScriptEase Engine.

ParameterOffset - The offset of the argument you are trying to access starting at 0. Variables are passed from left to right.

RETURN Returns a jseVariable pointer if a valid index is given. Otherwise returns *NULL*. If index is invalid then the error handling routines will have been called.

SEE ALSO jseFuncVarCount, jseGetFunction, jseFuncVarNeed

jseFuncVarCount

DESCRIPTION Get the number of parameters passed to a wrapper function.

SYNTAX
```
uint
jseFuncVarCount(jseContext jsecontext);
```

COMMENTS This function determines how many arguments were passed to a ScriptEase function.

jseContext - The context for this wrapper function. Use the value supplied to the wrapper function by the ScriptEase Engine.

RETURN The number of arguments passed to this wrapper function.

SEE ALSO jseFuncVar, jseGetFunction, jseFuncVarNeed

jseFuncVarNeed

DESCRIPTION Get a ScriptEase function wrapper argument and validate its type.

SYNTAX
```
jseVariable
jseFuncVarNeed(jseContext jsecontext,
               uint parameterOffset,
               jseVarNeeded need );
```

COMMENTS This function is used to access function arguments to a ScriptEase wrapper function. It returns a jseVariable pointer, and does type checking and possible conversion.

jseContext - The context for this wrapper function. Use the value supplied to the wrapper function by the ScriptEase Engine.

parameterOffset - The offset of the argument you are trying to access, starting at 0.

need - The type of the argument you are trying to access. It can be one or more of the following values. If you are supplying two possible types, they should be OR'ed together.

JSE_VN_UNDEFINED get an undefined variable.

JSE_VN_NUMBER get a number.

JSE_VN_NULL get a *NULL* variable.

JSE_VN_STRING get a string or byte array.

JSE_VN_BOOLEAN get a boolean variable.

JSE_VN_INT get a number that can be represented as a long with no loss of precision.

JSE_VN_FUNCTION get a function object.

JSE_VN_BYTE get a number that can be represented as a byte with no loss of precision.

JSE_VN_BUFFER get a buffer.

JSE_VN_OBJECT get an object.

JSE_VN_ANY accepts any variable type.

JSE_VN_NOT() accepts any variable not passed as a parameter. For example:

JSE_VN_NOT(JSE_VN_NUMBER | JSE_VN_STRING)

will accept any variable that is not a number or a string.

JSE_VN_CONVERT(from, to) This macro converts variables of the type indicated by the first parameter to the type indicated by the second parameter. For example:

JSE_VN_CONVERT(JSE_VN_ANY, JSE_VN_STRING)

will convert any type of variable received to a string. You cannot convert from JSE_VN_INT, JSE_VN_BYTE, or JSE_VN_FUNCTION.

JSE_VN_COPYCONVERT This option indicates that if a variable must be converted (with JSE_VN_CONVERT() or with the jseOptLenientConversion option), a copy of the variable will be made and converted, so that the original variable retains its type and value. You may also use the macro JSE_FUNC_VAR_NEED(). If the index or type are invalid this macro will not return and the scripting session will be terminated.

JSE_VN_CREATE - create variable for explicit jseDestroyVariable.

JSE_VN_READ - variable is for reading only.

JSE_VN_WRITE - variable is for writing only.

RETURN returns a jseVariable pointer if a valid index is given and the type specified is found. Otherwise returns *NULL*. If index is invalid or the type is incorrect, an error message will have been called, and you should return from the function.

SEE ALSO jseFuncVar, jseGetFunction, jseFuncVarCount

jseGarbageCollect

DESCRIPTION jseGarbageCollect allows you to control ScriptEase's internal garbage collection. This API call allows you to force an immediate collection or prevent collections from occurring.

SYNTAX
```
void
jseGarbageCollect(jseContext jsecontext,uint action);
```

PARAMETERS jseContext - The current executing context.

action - The garbage collection option to apply

COMMENTS The action may be one of three possible values:

JSE_GARBAGE_COLLECT - perform a garbage collection immediately, even if you have turned off garbage collection using this call.

JSE_GARBAGE_OFF - turn off garbage collecting. This increments a count, so if you turn it off more than once, you will need to turn it on more than once. When garbage collection is turned off, the engine will allocate more memory when it runs out rather than using collection to free up unused memory. JSE_GARBAGE_ON - turn garbage collection back on.

Please note that turning off garbage collection will significantly slow execution in addition to using a lot more memory; almost all programs need never do this.

Forcing a collection is useful because objects that have destructors and are freed will have their destructors called. This will allow you to ensure all such destructors have been called at a particular point in your program.

jseGetArrayLength

DESCRIPTION Get the span of elements in a ScriptEase variable object, string or buffer.

SYNTAX
```
ulong
```

```
jseGetArrayLength(   jseContext jsecontext,
jseVariable variable,   slong  *MinIndex );
```

COMMENTS This routine determines the size (length) of a ScriptEase object, string or buffer.

jseContext - The current executing context.

variable - array variable for which to check the span.

MinIndex - When the function returns this will be set to the index value of the first element in the ScriptEase array. This value will not be greater than zero.

In evaluation objects, this function will only consider elements with numeric indices.

For example, with this code the length of "foo" is 4:

```
var foo= new object();
foo[3] = "hello"
foo blah = "goodbye
```

RETURN The length of the array. This will be zero or greater.

SEE ALSO jseCreateVariable, jseCreateSiblingVariable, jseCreateLongVariable, jseDestroyVariable, jseSetArrayLength

jseGetAttributes

DESCRIPTION Get a variable's attributes.

SYNTAX

```
jseVarAttributes
jseGetAttributes(    jseContext jsecontext,
jseVariable variable );
```

COMMENTS This function is used to access the data associated with a jseTypeByte variable.

jseContext - The current executing context.

variable - The ScriptEase variable to read.

RETURN The attributes assigned to variable.

SEE ALSO jseSetAttributes

jseGetBoolean

DESCRIPTION Get boolean from a jseVariable.

SYNTAX
```
jsebool
jseGetBoolean(    jseContext jsecontext,
jseVariable variable );
```

COMMENTS This function retrieves the data associated with a jseTypeBoolean variable.

jseContext - The current executing context.

variable - The ScriptEase variable to read.

RETURN The attributes assigned to variable.

SEE ALSO jseGetAttributes

jseGetBuffer

DESCRIPTION Get buffer data from a jseVariable.

SYNTAX
```
void _HUGE_ *
jseGetBuffer(jseContext jsecontext,
             jseVariable variable,
             ulong *filled);
```

COMMENTS Get buffer data from a jseVariable. Buffer data can have binary and *NULL* characters in the block and although it will always be *NULL* terminated, the final *NULL* is not considered part of the data and is not part of the length. The returned data can not be modified.

jseContext - The current executing context.

variable - The jseVariable for the buffer being accessed.

filled - This variable will be set to the length of the data buffer on return.

RETURN The buffer data.

SEE ALSO jseGetWritableBuffer, jseGetString

jseGetByte

DESCRIPTION Get the unsigned-byte value of a numeric variable.

SYNTAX
```
uchar
jseGetByte(jseContext jsecontext,
           jseVariable variable );
```

COMMENTS This function gets the data associated with a jseTypeByte variable.

jseContext - The current executing context.

variable - The ScriptEase variable to read.

RETURN The value contained in the numeric variable as a byte (i.e. uchar).

SEE ALSO jsePutByte

jseGetCurrentThisVariable

DESCRIPTION Get the current "this" variable.

SYNTAX
```
jseVariable
jseGetCurrentThisVariable(jseContext jsecontext);
```

COMMENTS This function is used to get the current "this" variable.

jseContext - The current executing context.

RETURN Returns a pointer to the current "this" variable.

SEE ALSO jseGlobalObject

jseGetExternalLinkParameters

DESCRIPTION Get a pointer to the external link parameters.

SYNTAX
```
struct jseExternalLinkparameters  *
jseGetExternalLinkparameters(jseContext jsecontext);
```

COMMENTS Use this function to get a pointer to the external link parameters. Use the pointer to temporarily change the call- back functions. It is up to you to save and restore this data structure.
jseContext - The current executing context.

RETURN A pointer to a jseExternalLinkparameters structure.

SEE ALSO jseInitializeExternalLink, jseTerminateExternalLink

jseGetFileNameList

DESCRIPTION This function returns a list of all files opened by the script.

SYNTAX
```
char * *
jseGetFileNameList(jseContext jsecontext,
                   int *number);
```

COMMENTS **jseContext** - The current executing context.

number - When the function returns this variable will be set to the number of currently open files.

RETURN An array of strings representing the file names of currently open files. Do not free or write this data.

jseGetFunction

DESCRIPTION Get a pointer to a ScriptEase variable.

SYNTAX
```
jseVariable
jseGetFunction(jseContext jsecontext,
   jseVariable object,
   const jsechar  *functionName,
   jsebool errorIfNotFound);
```

COMMENTS This function gets a pointer to a given ScriptEase Library function, or any other function in the script being executed.

jseContext - The context to search for the given function name.

object - The object the function will be associated with. Use *NULL* to associate the function with the global object.

functionName - A string containing the name of the ScriptEase function you are searching for.

errorIfNotFound - If this flag is set to *True*, an error message will be displayed if the requested function can not be found.

RETURN A jseVariable for the requested function. This function will cause a temporary variable to be freed when the current context is ended, such as when returning from a wrapper function. To avoid the temporary variable (e.g. not calling from a wrapper or calling frequently) use jseMemberExec(...jseCreateVariable) and test that the variable is a function with jseIsFunction()
This function will return *NULL* if the requested function wasn't found.

SEE ALSO jseCallFunction, jseCurrentFunctionName

jseGetIndexMember

DESCRIPTION Get a jseVariable pointer to a numerically indexed object property.

SYNTAX
```
jseVariable
jseGetIndexMember(jseContext jsecontext,
                  jseVariable objectVariable,
                  slong index);
```

COMMENTS This routine gets a jseVariable pointer to an object property. This function is intended for use with the numbered properties of objects. To get a property that is named with a string, use jseGetMember().

jseContext - The current executing context.

objectvariable - The jseVariable pointer to the object from which to get a property.

index - The index of the desired property.

RETURN jseVariable is returned or *NULL* if the index is invalid.

SEE ALSO jseGetNextMember, jseIndexMember, jseMember, jseDeleteMember, jseGetIndexMemberEx

jseGetIndexMemberEx

DESCRIPTION Get a jseVariable pointer to a numerically indexed object property.

SYNTAX
```
jseVariable
jseGetIndexMember(jseContext jsecontext,
                  jseVariable objectVariable,
                  slong index uword16 flags);
```

COMMENTS This routine gets a jseVariable pointer to an object property. This function is intended for use with the numbered properties of objects. To get a property that is named with a string, use jseGetMemberEx().

jseContext - The current executing context.

objectvariable - The jseVariable pointer to the object from which to get a property.

index - The index of the desired property.

flags - this should be set to one of the following:

jseCreateVar - the variable must be explicitly destroyed with jseDestroyVariable() when you are done with it. If this flag is not specified then the variable is put in a temporary list to be destroyed when the current context finishes, such as when returning from a wrapper function.

jseDefault - the variable will be freed when the function exits.

JSE_VN_LOCKREAD

JSE_VN_LOCKWRITE

RETURN jseVariable is returned or *NULL* if the index is invalid.

SEE ALSO jseGetNextMember, jseIndexMember, jseMember, jseIndexMemberEx, jseMemberEx, jseGetIndexMember, jseGetMember, jseGetIndexMemberEx, jseGetMemberEx, jseDeleteMember

jseGetLastApiError

DESCRIPTION Retrieve error message set by failed API call.

SYNTAX
```
const jsechar *
jseGetLastApiError()
```

COMMENTS If one of the API calls fails, it will set an error message explaining the reason for the failure. This function retrieves this error message. You may also use the macro jseApiOK to determine whether the error flag has been set; this macro returns *True* if there is an error message set and *False* if there is not.

RETURN returns a pointer to a string containing the error message.

jseGetLinkData

DESCRIPTION Get the optional data associated with a jseContext.

SYNTAX
```
void  *
jseGetLinkData(jseContext jsecontext );
```

COMMENTS This function gets a pointer to the user supplied data for a given context, i.e., the data supplied in the linkData parameter to jseInitializeExternalLink().

jseContext - The current executing context

RETURN Far pointer to the user supplied data for the given context.

SEE ALSO jseInitializeExternalLink, jseTerminateExternalLink, jseGetExternalLinkparameters

jseGetLong

DESCRIPTION Get the long value of a numeric variable.

SYNTAX

```
slong
jseGetLong(jseContext jsecontext,
           jseVariable variable );
```

COMMENTS Use this function to access the data of a jseTypeNumber variable, cast to an slong.

jseContext - The current executing context.

variable - The ScriptEase variable to read.

RETURN The value contained in the numeric variable as an slong.

SEE ALSO jsePutLong

jseGetMember

DESCRIPTION Get a jseVariable pointer to a ScriptEase object property.

SYNTAX

```
jseVariable
jseGetMember(jseContext jsecontext,
             jseVariable objectVariable,
             const jsechar  *Name );
```

COMMENTS This routine gets a jseVariable pointer to an object property.

jseContext - The current executing context.

objectVariable - The jseVariable pointer to the object from which to get a property. Use *NULL* to indicate the global variable. The prototype will be searched.

Name - The name of the object property.

RETURN A jseVariable pointer to the requested object property, or *NULL* if the object does not have the specified property..

SEE ALSO jseGetNextMember, jseMember, jseDeleteMember

jseGetMemberEx

DESCRIPTION Get a jseVariable pointer to a ScriptEase object property.

SYNTAX

```
jseVariable
jseGetMemberEx(jseContext jsecontext,
               jseVariable objectVariable,
               const jsechar  *Name,
               uword16 flags );
```

COMMENTS This routine gets a jseVariable pointer to an object property.

jseContext - The current executing context.

objectVariable - The jseVariable pointer to the object from which to get a property. Use *NULL* to indicate the global variable.

Name - The name of the object property.

flags - this should be set to one of the following:

jseCreateVar - the variable must be explicitly destroyed with jseDestroyVariable() when you are done with it. If this flag is not specified then the variable is put in a temporary list to be destroyed when the current context finishes, such as when returning from a wrapper function.

jseDefault - the variable will be freed when the function exits.

RETURN A jseVariable pointer to the requested object property, or *NULL* on failure.

SEE ALSO jseMemberEx, jseGetNextMember, jseMember, jseMemberEx, jseIndexMember, jseIndexMemberEx, jseGetIndexMember, jseGetIndexMemberEx, jseDeleteMember

jseGetNextMember

DESCRIPTION Routine to enumerate all the properties of an object.

SYNTAX

```
jseVariable
jseGetNextMember(jseContext jsecontext,
                 jseVariable objectVar,
                 jseVariable prevMemberVariable,
                 const jsechar  *  * name );
```

COMMENTS This function allows you to get all the properties of a ScriptEase object variable by stepping through them one at a time. It isn't necessary to know the names of the properties. In the first call, *NULL* is provided as the previous property; the first property of the object will be returned. This function will return properties which have the dontEnum attribute set.

jseContext - The current executing context.

objectVar - The jseVariable pointer to the object from which to retrieve properties. Use *NULL* to indicate the global variable.

prevMemberVariable - Pointer to the previous object property, if this is set to *NULL*, the first member will be returned.

name - On return, the name of the object property that was returned. Do not alter this variable.

RETURN A jseVariable pointer to the next object property. This value should be used on subsequent calls to retrieve the next properties. When *NULL* is returned, there are no more object properties.

SEE ALSO jseGetMember, jseMember, jseDeleteMember

jseGetNumber

DESCRIPTION Get the floating-point numeric value of a numeric variable.

SYNTAX

```
jseGetNumber
jseGetNumber(jseContext jsecontext,
             jseVariable variable );
```

COMMENTS Use this function to access the data of a jseTypeNumber variable

jseContext - The current executing context.
variable - The ScriptEase variable to read.

RETURN The value contained in the numeric variable as a jseNumber (i.e. floating-point number).

SEE ALSO jsePutLong, jsePutNumber, jseGetLong, jseGetByte

jseGetString

DESCRIPTION Get string data from a ScriptEase variable.

SYNTAX

```
const jsechar *
jseGetString(jseContext jsecontext,
             jseVariable variable
             ulong * filled);
```

COMMENTS Get string data from a variable. The returned data must not be modified.

jseContext - The current executing context.

variable - The ScriptEase variable to read.

filled - When the function returns this will be set to the length of the string.

RETURN The data will be *NULL*-terminated, but this terminating null character is not considered part of the variable and not considered when determining the variable length. Note also that ECMAScript strings may contain embedded *NULL*s.

SEE ALSO jseGetBuffer, jseGetWritableString, jseGetWritableBuffer, jseCopyString, jseCopyBuffer, jsePutString, jsePutBuffer

jseGetType

DESCRIPTION Get the jseType of a jseVariable.

SYNTAX

```
jseDataType
jseGetType(jseContext jsecontext,
           jseVariable variable );
```

COMMENTS This function is used to determine the specified jseVariable's type.

jseContext - The current executing context.

variable - The jseVariable whose type is being checked.

RETURN The type of the variable passed as the argument. Valid types are jseTypeUndefined, jseTypeNull, jseTypeNumber, jseTypeString, and jseTypeBuffer, jseTypeObject, jseTypeBoolean.

SEE ALSO jseConvert, jseAssign

jseGetVariableName

DESCRIPTION Get the name of a script variable corresponding to the given jseVariable.

SYNTAX

```
jsebool
jseGetVariableName(jseContext jsecontext,
                   jseVariable variableToFind,
                   char * const buffer
                   ulong bufferSize);
```

COMMENTS This function gets the name of the variable corresponding to variableToFind. For example, if there is an error in executing the script and you wish to inform the user that a variable is of the wrong type, you can use this function to get the name of the variable as it is referred to in the script.

RETURN *True* if successful, *False* if the variable was not found

SEE ALSO jseGetType, jseGetFunctionName

jseGetWriteableBuffer

DESCRIPTION Get buffer data from a jseVariable.

SYNTAX

```
void _HUGE_ *
jseGetWriteableBuffer(jseContext jsecontext,
                      jseVariable variable,
                      ulong *filled);
```

COMMENTS Get buffer data from a variable. Buffer data can have binary and `'\0'` characters in the block. The returned data can be modified. The filled parameter will be assigned the length of the data buffer.

jseContext - The current executing context. **variable** - The jseVariable handle to the buffer being accessed.

filled - This variable will be set to the length of the data buffer on return.

RETURN The buffer data.

SEE ALSO jseGetBuffer, jseGetString

jseGetWriteableString

DESCRIPTION Get string data from a ScriptEase variable.

SYNTAX

```
jsechar *
jseGetWriteableString(jseContext jsecontext,
                      jseVariable variable,
                      ulong *filled)
```

COMMENTS Get string data from a ScriptEase variable. This function differs from jseGetString in that the returned data can be modified. However, to make any changes reflected in the variable, you must use jsePutString() to update the variable. This function merely provides a buffer you can modify.

jseContext - The current executing context.

variable - The jseVariable handle to the string variable being accessed.

filled - This variable will be set to the length of the string.

RETURN The string data contained in variable.

SEE ALSO jseGetString, jseGetNumber

jseGlobalObject

DESCRIPTION Get the current global object.

SYNTAX

```
jseVariable
jseGlobalObject(jseContext jsecontext);
```

COMMENTS This function is used to get the current global object.

jseContext - The current executing context.

RETURN Returns a pointer to the current global object.

SEE ALSO jseGetCurrentThisVariable

jseIndexMember

DESCRIPTION Retrieve a numerically indexed variable from an object; create it if it does not exist.

SYNTAX

```
jseVariable
jseIndexMember(jseContext jsecontext,
               jseVariable objectVar,
               slong index,
               jseDataType type);
```

COMMENTS This function is intended to get the numbered properties of objects. To get named properties, use jseMember().

jseContext - The current executing context.

objectVar - The array object to query.

index - The index of the variable to retrieve.

type - The type of the desired variable.

RETURN The desired variable. If it does not exist it will be created.

SEE ALSO jseTerminateEngine

jseIndexMemberEx

DESCRIPTION Retrieve a variable from a numerically-indexed object; create it if it does not exist.

SYNTAX

```
jseVariable
jseIndexMemberEx(jseContext jsecontext,
                 jseVariable objectVar,
                 JSE_POINTER_SINDEX index,
                 jseDataType type
                 uword16 flags);
```

COMMENTS This function is intended to get the numbered properties of objects. To get named properties, use jseMemberEx().

jseContext - The current executing context.

objectVar - The array object to query.

index - The index of the variable to retrieve.

type - The type of the desired variable.

flags - this should be set to one of the following:

jseCreateVar - the variable must be explicitly destroyed with jseDestroyVariable() when you are done with it

jseDefault - the variable will be freed when the function exits.

JSE_VN_LOCKREAD

JSE_VN_LOCKWRITE

RETURN The desired variable. If it does not exist it will be created.

SEE ALSO jseTerminateEngine

jseInitializeEngine

DESCRIPTION This call initializes the ScriptEase Interpreter Engine.

SYNTAX
```
uint
jseInitializeEngine();
```

COMMENTS Call this before any other call in the toolkit to initialize the processor.

RETURN Returns the ID of the engine for version number verification.

SEE ALSO jseTerminateEngine

jseInitializeExternalLink

DESCRIPTION Routine to initialize a ScriptEase context.

SYNTAX
```
jseContext
jseInitializeExternalLink(void _FAR_ *linkData,
                jseExternalLinkparameters  *linkParms
                   const char * globalVarName,
                cost char  *accessKey);
```

COMMENTS **LinkData** - this variable contains any global data for your scripting session. The interpreter ignores this data, but since it is included in the jseContext it is available to all functions registered with it. This is a "cookie" which provides you with a way to pass information to the routines you supply to a ScriptEase context. The exact pointer you supply will be available to all functions called by the ScriptEase Engine for the ScriptEase context being created. Use LinkData to maintain global information throughout your scripting session; or to store any values your script will use later on. The ScriptEase engine itself does not modify link data.

linkParms - this structure (jseExternalLinkparameters) contains the user defined properties of the ISDK. They are described in full below.

globalVarName - this parameter, a string, is the name you wish to give the global object.

accessKey - this is the key (supplied by Nombas) needed to activate your copy of ScriptEase:Integration SDK.

The jseExternalLinkparameters structure has this prototype:

```
struct jseExternalLinkparameters
{
  jseFileFindFunc FileFindFunc;
  jseErrorMessageFunc PrintErrorFunc;
  jseMayIContinueFunc MayIContinue;
  jseGetSourceFunc  GetSourceFunc;
  jseAppLinkFunc  AppLinkFunc;
  const char *jseSecureCode;
  uword32 options
};

typedef
jsebool( FAR_CALL *jseFileFindFunc )
      (
         jseContext jsecontext,
       char * FileSpec,
       char * FilePathResults,
       uint FilePathLen,
       jsebool FindLink
        );

typedef
void ( FAR_CALL *jseErrorMessageFunc)
(
        jseContext jsecontext,
      const char  *ErrorString
       );

typedef
jsebool ( FAR_CALL *jseMayIContinueFunc)
       (jseContext jsecontext);

typedef
jsecontext( FAR_CALL *jseAppLinkFunction)
         (
            jseContext jsecontext,
          const char  *Errorstring
           );
typedef
jsecontext( FAR_CALL *jseAppLinkFunction)
     (
        jseContext jsecontext,    const char
*Errorstring
       );

typedef
jseGetSourceFunc(jseContext jsecontext,
               struct jseToolkitAppSource *
                 toolkitAppSource,
          jseToolkitAppSourceFlags flags);
```

jseFileFindFunc - Function that will be called every time a file needs to be opened by the interpreter. Use this function to provide the full path and name to a file that exists. This may be set to *NULL* if it is not used.

jseErrorMessageFunc - Function to be called by the interpreter when a script encounters an error condition. If this parameter is set to *NULL*, the default handler is used.

jseMayIContinueFunc - Function to be called before executing every command in a script for permission to continue interpreting it. This allows for implementing single step debugging environments. If this parameter is set to *NULL*, the ScriptEase engine will not call any routine.

To use this function to provide a debugging interface, use jseLocateSource() to retrieve the name and line number of the current location in the script being run. This will provide a callback monitoring function for your scripts, call multitasking tickler routines, or a check on external status such as the pressing of ctrl-C or break.

```
        typedef jsebool (JSE_CFUNC
*jseMayIContinueFunc )
                          (jseContext jseContext);
```

jseSecureCode - Either a full file name and path or a block of JavaScript code that performs the security checking. Set this parameter to *NULL* if no security checking is needed.

jseGetSource - This function tells the interpreter how to open files included with #include statements and scripts passed to jseInterpret as files. The third parameter passed to this function is a flag with one of the following values:

jseNewOpen - open and initialize the file for reading.

jseNextLine - retrieve next line of code from file.

jseClose - close file and cleanup.

jseAppLinkFunction - This callback function can be used to let your application create a new context initialized with globals, libraries, #defines, etc. A common use would be to create a new context in a new thread. Set to *NULL* if you do not use it.

Options - this is an or mask of the following flags. They define how the interpreter treats variables.

jseDefault - Use this flag to use the system defaults.

jseOptRequireVarKeyword - Use this flag if you want to force your users to use the 'var' keyword when creating variables.

jseOptRequireFunctionKeyword - Use this flag if you want to force your users to use the 'function' keyword when creating functions.

jseOptDefaultLocalVars - Use this flag if you want variables declared in a local environment to be local, regardless of whether the var keyword is used or not. (In JavaScript, variables declared without the var keyword would normally be global). If there is a like-named global variable, instead of creating a local variable the global variable would be used.

jseOptDefaultCBehavior - If this flag is defined, functions will be treated as if they were created with the 'cfunction' keyword, regardless of what keyword they were defined with.

jseOptWarnBadMath - If this flag is set, the interpreter will notify you when you make illegal mathematical calculations (such as dividing by zero). In JavaScript, dividing by zero normally returns the value *NaN* and does not generate an error.

jseOptLenientConversion - this option causes variables to automatically be converted to the required type if possible, instead of generating an error. With this option set the macro JSE_VN_CONVERT() will always behave as if the first parameter passed were JSE_VN_ALL. The jsePutxxx() functions will convert the variable to the required type. If you are retrieving data from a variable, if the variable is not of the correct type a copy of the variable will made, converted to the correct type, and returned.

jseOptIgnoreExtraParameters - If this option is set, the interpreter will ignore any parameters greater than the maximum allowed for the function (specified in the Function Descriptor table added to the context with jseAddLibrary().

RETURN returns a jseContext initialized with the values provided.

SEE ALSO jseGetExternalLinkparameters()

---

jseInterpret

DESCRIPTION Interpret a ScriptEase script.

SYNTAX

```
jsebool
jseInterpret(jseContext jsecontext,
   const  * sourceFile,
   const  * sourceText,
   const void * pretokenizedBuffer,
   jseNewContextSettings jseNewContextSettings,
   int howToInterpret,
   jseContext localVariableContext,
   jseVariable  *returnVar);
```

COMMENTS This call is the heart of the ScriptEase engine. After your ScriptEase toolkit environment is set up, call this routine to interpret scripts.

jseContext - The context to use for this interpret.

sourceFile - This argument is a string of the filename and path to a JavaScript file or *NULL* if you are interpreting JavaScript source from memory.

sourceText - This argument is either a block of SE code in memory to interpret or if interpreting code from a file, the optional arguments to pass to the script. If you do not need to use this parameter it should be set to *NULL*.

pretokenizedBuffer - If you are interpreting code that has been pretokenized with the jseCreateCodeTokenBuffer(), the code should go here. Otherwise, set this parameter to *NULL*.

jseNewContextSettings - These flags specify which elements of the jseContext about be created will be created new. Otherwise the elements will be inherited from the current jseContext. Use one or more of the following flags OR'ed together:

jseNewNone - Do not create any new elements.

jseNewFunctions - Create new functions.

jseNewSecurity - Reinitialize security before interpreting the script.

jseAllNew - Create new elements for all categories (functions will be inherited from the parent jseContext).

howToInterpret - A flag to specify the method of interpretation. Use one or more of the following, joined by a bitwise or (|):

JSE_INTERPRET_NO_INHERIT - This flag prevents global variables from being passed to the new jseContext.

JSE_INTERPRET_CALL_MAIN - Call main() after running initialization code.

localVariableContext - This parameter is a jseContext or *NULL*. If you are calling jseInterpret from within a wrapper function, pass jsePreviousContext(jsecontext); otherwise pass *NULL*.

returnCode - If the function executes successfully (i.e., returns *True*), on return this will contain the value returned by the JavaScript being executed. This variable must later be destroyed with jseDestroyVariable(). If you don't need to use this value, pass in *NULL*. The return value will be cleaned up automatically.

RETURN *True* if the script was successfully executed, *False* if not.

jseInterpExec

DESCRIPTION Interpret a ScriptEase script

SYNTAX
```
jseContext
jseInterpretExec(jseContext jsecontext);
```

COMMENTS **jseContext** - The context to use for this interpret.
See jseInterpInit() for a description on using this function.

RETURN The context to pass to the next call to this function. *NULL* indicates the script is done executing.

jseInterpInit

DESCRIPTION Interpret a ScriptEase script one statement at a time under your program's control.

SYNTAX
```
jseContext
jseInterpret(jseContext jsecontext,
   const  * sourceFile,
   const  * sourceText,
   const void * pretokenizedBuffer,
   jseNewContextSettings jseNewContextSettings,
   int howToInterpret,
   jseContext localVariableContext,
   jseVariable  *returnVar);
```

COMMENTS jseInterpInit(), jseInterpExec() and jseInterpTerm() provide an alternative to jseInterpret() for interpreting scripts. The two systems work in slightly different ways. jseInterpret() will call the MayIContinueFunc() defined in the jseContext before each script line is executed.

With jseInterpInit() et al, you have more control over how the script executes. jseInterpInit() initializes the script for interpretation. It takes the same parameters as jseInterpret(). jseInterpInit() returns a new jseContext for the script, which is then passed to jseInterpExec().

The script is executed through repeated calls to jseInterpExec() taking this jseContext() as its only parameter and returning an updated jseContext that must be passed again to jseInterpExec to execute successive lines. If there are no more lines to execute, jseInterpExec() returns *NULL*. MayIContinueFunc() will not be called.

When jseInterpExec() returns *NULL*, the script has completed, and you should call jseInterpTerm() to clean up the interpret. jseInterpTerm() takes one parameter, the original jseContext passed to jseInterpInit() and not one of jseContexts returned from jseInterpInit() or jseInterpExec(). See jseInterpret for a description of parameters.

Note that jseInterpInit() is only provided so that you can execute your script one statement at a time under your program's control. It's use is probably unneeded 99% of the time, as the MayIContinue function (defined in the jseExternalLinkParameters structure) can do what you need instead. In fact, there is NO functionality related to scoping that jseInterpInit() has that cannot be done using jseInterpret().

RETURN The context to use with jseInterpExec() the first time or *NULL* if some error prevented the interpreting from being initialized..

jseInterpTerm

DESCRIPTION Terminate a ScriptEase script interpretation session.

SYNTAX

```
jseVariable
jseInterpTerm(jseContext jsecontext,
              jsebool traperrors);
```

COMMENTS **jseContext** - The context to use for this interpret.
See jseInterpInit() for a description of using this function.

RETURN The variable returned as the result of the script. You must destroy it when you are done with it. *NULL* is returned if there was an error interpreting the script.

jseIsFunction

DESCRIPTION Test whether a variable is a script or wrapper function registered with the supplied jseContext.

SYNTAX

```
jsebool
jseIsFunction(jseContext jsecontext,
              jseVariable functionVariable );
```

COMMENTS This function tests whether functionVariable is a registered function or not. If functionVariable was retrieved from a call to jseGetFunction(), this test is not necessary.

jseContext - The current executing context.

functionVariable - The variable being tested.

RETURN *True* if functionVariable is a registered function; *False* if it is not.

SEE ALSO jseCreateWrapperFunction, jseIsLibraryFunction

jseIsLibraryFunction

DESCRIPTION Test whether a variable is a wrapper function registered with the supplied jseContext.

SYNTAX
```
jsebool
jseIsLibraryFunction(jseContext jsecontext,
                     jseVariable functionVariable );
```

COMMENTS This function tests whether functionVariable is a function added with jseAddLibrary or not.

jseContext - The current executing context.

functionVariable - The variable being tested.

RETURN Returns *True* if the function was added with jseAddLibrary(); otherwise returns *False*.

SEE ALSO jseCreateWrapperFunction, jseIsFunction

jseLibErrorPrintf

DESCRIPTION Prints a formatted string describing the error encountered and flags the interpreter to quit execution.

SYNTAX
```
void
jseLibErrorPrintf(jseContext exitContext,
                  CPP_CONST char * formatS,... );
```

COMMENTS The function sets the error flag for the jseContext and prints the format string. The string lets you supply information about why the error occurred. It uses the same arguments and format string as the standard printf function.

exitContext - The context to associate with the Error.

formatS - The format string for the printf statement, followed by any parameters it takes.

If an error condition has already been flagged, then this function performs no action.

RETURN None.

SEE ALSO jseLibSetErrorFlag, jseLibSetExitFlag

jseLibSetErrorFlag

DESCRIPTION Set the Lib Error flag.

SYNTAX
```
void
jseLibSetErrorFlag( jseContext jsecontext );
```

COMMENTS Use this function sets the lib error flag to indicate that an error condition exists. The script will be terminated and any necessary cleanup performed.

jseContext - The context to set the lib error flag for.

RETURN None.

SEE ALSO jseLibErrorPrintf, jseLibSetExitFlag

jseLibSetExitFlag

DESCRIPTION Set the ScriptEase Lib exit flag.

SYNTAX
```
void  jseLibSetExitFlag(jseContext jsecontext,
                        jseVariable exitVariable);
```

COMMENTS Sets exit flag for this jseContext and saves exit variable. The script will clean-up and exit on return from this wrapper function.

jseContext - The current executing context.

exitVariable - The value to be returned by the script. This is the variable returned in jseInterpret.

RETURN None.

SEE ALSO jseLibErrorPrintf, jseLibSetErrorFlag

jseLibraryData

DESCRIPTION Get the library data for the current library

SYNTAX
```
void _FAR_ *
jseLibraryData(   jseContext jsecontext );
```

COMMENTS This function gets the library data for a given library. It is intended for use within a wrapper function for a library.

jseContext - The jseContext for which to get the library data. Use the value passed to your wrapper function.

RETURN Far pointer to the requested library data, the value returned by jseLibraryInitFunction specified in jseAddLibrary() or add in jseCreateWrapper or jseMemberWrapper.

SEE ALSO jseAddLibrary

JseLocateSource

DESCRIPTION Get the file information for the currently running script.

SYNTAX
```
const jsechar  *
jseLocateSource(jseContext jsecontext,
                uint  *lineNumber );
```

COMMENTS Returns a pointer to the name of the source file for the code currently being executed, and sets **lineNumber** to the line number currently being executed or parsed. If there is no current file (as when interpreting a string) *NULL* will be returned.

jseContext - The context to use for this interpret.

lineNumber - Pointer to the current source file number.

RETURN A string containing the name of the source file for the currently executing code. Do not modify this string

jseMember

DESCRIPTION Get a ScriptEase variable reference to a ScriptEase structure element.

SYNTAX
```
jseVariable
jseMember(jseContext jsecontext,
          jseVariable objectVar,
          const jsechar  *Name,
          jseDataType DType)
```

COMMENTS This routine gets a ScriptEase variable reference for an object's property. Once the jseVariable reference is obtained, use the data access functions to get the data. If the variable does not exist, it will be created when it is read from or written to.

jseContext - The context to use for this interpret.

objectVar - The object to get a property from.

Name - The name of the object property.

DType - This argument specifies the type of object property variable that will be created if the variable does not already exist.

Note: this function has been deprecated in version 4.03. Internally it calls jseMemberEx() with the flags parameter set to jseDefault.

RETURN A jseVariable pointer to the requested object property, or *NULL* on failure. If the property does not exist it will be created. Failure means the interpreter ran out of memory.

SEE ALSO jseMemberEx, jseGetMember, jseGetMemberEx, jseIndexMember, jseIndexMemberEx, jseGetNextMember, jseDeleteMember, jseGetIndexMember, jseGetIndexMemberEx

jseMemberEx

DESCRIPTION Get a ScriptEase variable reference to a ScriptEase structure element.

SYNTAX

```
jseVariable
jseMemberEx(jseContext jsecontext,
            jseVariable objectVar,
            const jsechar  *Name,
            jseDataType DType   uword16 flags)
```

COMMENTS This routine gets a ScriptEase variable reference for an object's property. Once the jseVariable reference is attained, use the data access functions to get the data. If the variable does not exist, it will be created.

jseContext - The context to use for this interpret.

objectVar - The object to get a property from.

Name - The name of the object property.

DType - This argument specifies the type of object property variable that will be created.

flags - this should be set to one of the following:

jseCreateVar - the variable must be explicitly destroyed with jseDestroyVariable() when you are done with it or else an internal error will occur.

jseDefault - the variable will be freed when the function exits.

RETURN A jseVariable pointer to the requested object property, or *NULL* on failure. If the property does not exist it will be created. Failure means the interpreter ran out of memory.

SEE ALSO jseGetMember, jseGetNextMember, jseDeleteMember

jseMemberWrapperFunction

DESCRIPTION Attach a new object method to a wrapper function.

SYNTAX

```
jseVariable
jseMemberWrapperFunction(jseContext jsecontext,
   jseVariable objectVar
      const jsechar _FAR *functionName
      jseLibraryFunction funcPtr,
   sword8 minVariableCount,
   sword8 maxVariableCount,
   uword8 varAttributes,   uword8 funcAttributes,
   void _FAR_ * fData);
```

COMMENTS This routine creates a function variable as an object method.

jseContext - The current executing context.

objectVar - The object that the function is a method of. Use *NULL* to make it a global function.

functionName - is the name of your function in a script. It should be a string such as "GetString". Your users will refer to the function by this name.

funcPtr - is a pointer to the function called by the ScriptEase Interpreter Engine, i.e., the name of the wrapper function that corresponds to the function listed above.

RETURN If successful, this returns the jseVariable created. If there is not enough system memory to create the variable (extremely unlikely), *NULL* will be returned.

jsePreDefineNumber

DESCRIPTION #define a string alias for a ScriptEase number value.

SYNTAX
```
void
jsePreDefineNumber(jseContext jsecontext,
   const jsechar FAR_ *findString,
   jseNumber replaceL );
```

COMMENTS Use this routine to define a float for use by interpreted scripts. When parsing the ScriptEase source, any instance of findString (case sensitive) that might otherwise refer to a variable is replaced with the value for replaceL.

This use:
```
jsePreDefineNumber(jsecontext,"PI",3.1415927);
```
is similar to the script having this statement:
```
#define PI 3.1415927
```

jseContext - The context to add this define to.

findString - *NULL*-terminated string to match in source.

replaceL - Number to substitute for findString when parsing source.

You can use this function to override the #define statements in a script.

RETURN None.

SEE ALSO jsePreDefineLong, jsePreDefineString

jsePreDefineLong

DESCRIPTION #define a string alias for a long-integer value.

SYNTAX
```
void
jsePreDefineLong(jseContext jsecontext,
                 const jsechar FAR_ *FindString,
                 slong ReplaceL );
```

COMMENTS Use this routine to define a long for use by interpreted scripts. When parsing the ScriptEase source, any instance of FindString (case sensitive) that might otherwise refer to a variable is replaced with the integer value for ReplaceL.

This use:
```
  jsePreDefineLong(jsecontext,"MILLION",1000000);
```
is similar to the ScriptEase source having a statement such as:
```
  #define MILLION 1000000
```

jseContext - The context to add this define to.

FindString - *NULL*-terminated string to match in ScriptEase source.

ReplaceL - Integer to substitute for FindString when parsing ScriptEase source.

You can use this function to override the #define statements in a script.

RETURN None.

SEE ALSO jsePredefineNumber, jsePredefineString

jsePreDefineString

DESCRIPTION Define a JavaScript string value.

SYNTAX

```
void
jsePreDefineString(jseContext jsecontext,
   const jsechar FAR_ *findString,
   const jsechar  *replaceString );
```

COMMENTS Use this routine to define a string for use by interpreted scripts. When parsing the source, any instance of findString (case sensitive) that might otherwise refer to a variable is replaced with replaceString. This use:

```
jsePreDefineString(jsecontext,"VERSION_STR",
   "Version 1.2.4 Beta");
```

is similar to the source having a statement such as:

```
#define VERSION_STR     "Version 1.2.4 Beta"
```

jseContext - The current executing context to add this define to, probably the root context.

FindString - *NULL*-terminated string to match in source.

ReplaceString - String to substitute for findString when parsing source. The replace string may be any sequence. You can use this function in your application to override the #define statements in any script it runs.

#define is used for text-replacement only, i.e., before the script is interpreted, all instances of findString are replaced with "replaceString," and the resulting text is interpreted as ScriptEase code.

RETURN None.

SEE ALSO jsePredefineNumber, jsePredefineLong

jsePreviousContext

DESCRIPTION Retrieve the previous context.

SYNTAX

```
jseContext
jsePreviousContext(jseContext jsecontext);
```

COMMENTS Given the current context, jsePreviousContext will find the previous one. The previous context will be the one that represents the script function that called the current function.

jseContext - The current executing context.

RETURN The previous ScriptEase context, or *NULL* if there wasn't one.

jsePush

DESCRIPTION Push a jseVariable onto a jseStack.

SYNTAX

```
void
jsePush(jseStack jsestack,
        jseVariable var,
        jsebool destroyWhenFinished);
```

This function pushes a jseVariable onto the jseStack.

COMMENTS **stack** - The stack to receive the variable. **var** - The jseVariable to push onto the stack.

destroyWhenFinished - A boolean flag, specifying whether or not the jseVariable on the stack should be destroyed when the stack is destroyed. You only set this to *true* if you are responsible for destroying a variable and wish to get rid of this responsibility. For instance, if you used jseCreateVariable() to construct a variable to pass as a parameter. By telling this routine to destroy it when done, you no longer have to worry about destroying it yourself.

RETURN None.

SEE ALSO jseCreateStack, jseDestroyStack

jsePutBoolean

DESCRIPTION Put boolean data into a jseVariable.

SYNTAX

```
void
jsePutBoolean(jseContext jsecontext,
              jseVariable variable,
              jsebool value);
```

COMMENTS This function is used to write data to a jseTypeBoolean variable.

jseContext - The current executing context.

variable - The ScriptEase variable to write.

value - Value to set the variable to; use True or False

RETURN None.

SEE ALSO jseGetBoolean

jsePutBuffer

DESCRIPTION Put buffer data into a jseVariable.

SYNTAX

```
void  jsePutBuffer(jseContext jsecontext,
   jseVariable variable,
   const void _HUGE_ *data,
   ulong size);
```

COMMENTS This function writes data to a jseTypeBuffer variable. **jseContext** - The current executing context.

variable - The ScriptEase variable to write data to.

data - Pointer to buffer data.

size - Size of the data buffer, in bytes.

RETURN None.

SEE ALSO jseGetBuffer, jseGetWritableBuffer

jsePutByte

DESCRIPTION Write data to a variable as a byte.

SYNTAX

```
void
jsePutByte(jseContext jsecontext,
   jseVariable variable,
   ubyte byteValue);
```

COMMENTS This function is used to write data to a variable of jseTypeNumber.

jseContext - The current executing context.

variable - The ScriptEase variable to write.

byteValue - Value to which the variable is to be set.

RETURN None.

SEE ALSO jseGetNumber, jsePutNumber, jsePutLong

jsePutNumber

DESCRIPTION Write numeric data to a jseVariable.

SYNTAX

```
void
jsePutNumber(jseContext jsecontext,
   jseVariable variable,
   jseNumber number);
```

COMMENTS This function is used to write data to a jseTypeNumber variable.

jseContext - The current executing context.

variable - The ScriptEase variable to write to.

number - Value to which the variable is to be set.

RETURN None.

SEE ALSO jseGetNumber, jsePutLong, jsePutByte

jsePutLong

DESCRIPTION Write signed-long data to a jseVariable.

SYNTAX

```
void
jsePutLong(   jseContext jsecontext,   jseVariable
variable,   slong longvalue);
```

COMMENTS This function is used to write data to a jseTypeNumber variable.

jseContext - The current executing context.

variable - The ScriptEase variable to write.

longvalue - Value to set the variable to.

None.

RETURN

SEE ALSO jseGetLong

jsePutString

DESCRIPTION Write string to a jseVariable.

SYNTAX

```
void
jsePutString(jseContext jsecontext,
   jseVariable variable,
   const jsechar _HUGE_ *data);
```

COMMENTS This function writes string data to a jseTypeString variable. The length of the string will be assumed to be the number of characters before the first `'\0'` character. If you wish to explicitly pass a string length, use jsePutStringLength().

jseContext - The current executing context.

variable - The ScriptEase variable to write.

data - Value to set the variable to.

RETURN None.

SEE ALSO jsePutStringLength, jseGetString, jseGetWritableString

jsePutStringLength

DESCRIPTION Write string to a ScriptEase variable.

SYNTAX

```
void
jsePutStringLength(jseContext jsecontext,
   jseVariable variable,
   const jsechar _HUGE_ *data,
   ulong size);
```

COMMENTS This function writes string data to a jseTypeString variable.

jseContext - The current executing context.

variable - The ScriptEase variable to write.

data - Value to set the variable to.

size - The length of the string in data.

RETURN None.

SEE ALSO jsePutString, jseGetString, jseGetWritableString

jseQuitFlagged

DESCRIPTION Check if current context has been flagged to terminate execution.

SYNTAX
```
uint
jseQuitFlagged(jseContext jsecontext );
```

COMMENTS **jseContext** - The current executing context to use for this interpret. returns 0 (*False*) if a call has not been made on this context to Exit (jseLibSetExitFlag()), or to report an error via any of the error reporting functions (jseLibSetErrorFlag() or jseLibErrorPrintf()). It is not necessary to call these functions after the jseXXX library functions, which include error status (if applicable) in their return codes.

This function can be valuable during debugging (e.g., in assert() statements) to ensure that the jseContext is valid. If you add functions that may set the error or exit flags and that don't indicate this information in their return codes, or if you are not checking return codes in some sections, then jseQuitFlagged() may be used.

Another use for this function is the case where your context may be handled in a callback, so you can save the context in a global and check later if there was a problem.

If your script should exit due to an exit flag or due to an error, then this function will return one of the following non-0 (non-*False*) values:

```
JSE_CONTEXT_ERROR       // ERROR flag set
JSE_CONTEXT_EXIT        // EXIT flag set
```

RETURN *(0) False* if this context is not flagged for exit due a call to jseLibSetExitFlag() or to an error call, else return reason for exit, indicated by one of the values described above.

SEE ALSO jseLibSetErrorFlag, jseLibSetExitFlag, jseLibErrorPrintf

jseReturnNumber

DESCRIPTION Return a number from a ScriptEase wrapper function.

SYNTAX

```
void
jseReturnNumber(jseContext jsecontext,
                jseNumber number);
```

COMMENTS This function is used to return a numeric value from a ScriptEase wrapper function. If you call any of the jseReturnXXX() functions again, the last call takes precedence. It creates a variable of type jseTypeNumber, assigns the **number** to it, and makes that the return from the wrapper function. It is not like 'exit()' in that your wrapper function continues executing. Typically, a call to this function is the last thing your wrapper function does before returning.

jseContext - The current executing context for this wrapper function. Use the value supplied to the wrapper function by the ScriptEase Engine.

number - The numeric value to return.

RETURN None.

jseReturnLong

DESCRIPTION Return a long from a ScriptEase wrapper function.

SYNTAX

```
void
jseReturnLong(jseContext jsecontext,
              slong longValue);
```

COMMENTS This function is used to return a long value from a ScriptEase wrapper function. If you call any of the jseReturnXXX() functions again, the last call takes precedence. It creates a variable of type jseTypeNumber, assigns the longValue to it, and makes that the return from the wrapper function. It is not like 'exit()' in that your wrapper function continues executing. Typically, a call to this function is the last thing your wrapper function does before returning.

jseContext - The current executing context. Use the value supplied to the wrapper function by the ScriptEase Engine.

longValue - The long value to return.

RETURN None.

SEE ALSO jseGetLong

jseReturnVar

DESCRIPTION Returns a jseVariable from a wrapper function.

SYNTAX

```
void
jseReturnVar(jseContext jsecontext,
   jseVariable variable,
    jsereturnAction retAction );
```

COMMENTS This function is used to generate a return value from a ScriptEase wrapper function. It will return the specified ScriptEase variable. If you call any of the jseReturnXXX() functions more than once, the last call takes precedence.

jseContext - The context executing context. Use the value supplied to the wrapper function by the ScriptEase Engine.

variable - The variable to be returned from this function.

retAction - Specifies how the variable to be returned shall be treated once you are done using it. The return action can be one of the following values:

jseRetTempVar - This is variable you own and are expected to delete. By passing it along using this flag, you no longer have to delete it. You have passed ownership to the system and it will delete it when it is finished with it.

jseRetCopyToTempVar - Create a new variable, copy to that variable (with jseAssign()), and then return that new variable to be destroyed when it is popped. Don't return this variable; return the copy. If you own this variable and are expected to delete it, you still must do so.

jseRetKeepLVar - This is similar to jseRetCopyToTempVar in that you still own the variable and must delete if appropriate. It differs in that no copy is made. If you change the variable (such as with jseConvert()), the change will be reflected in the value returned from the function.

RETURN None.

jseSetArrayLength

DESCRIPTION Set the length of a string, buffer or numerically-indexed object.

SYNTAX

```
void
jseSetArrayLength(jseContext jsecontext,
                  jseVariable variable,
                  slong MinIndex,
                  ulong length);
```

COMMENTS This routine sets the length of a ScriptEase string, buffer or numerically-indexed object. This function will create new array entries if they are needed, and destroy those that are no longer needed, i.e., that are outside of the bounds of the new array.

jseContext - The current executing context.

variable - the ScriptEase variable for which the length will be set.

MinIndex - the index value to use for the first element of the array. Must be less than or equal to zero.

length - length of the string or buffer, or one greater than the maximum numerically indexed property or an object. Must be greater than or equal to zero.

RETURN None.

SEE ALSO jseGetArrayLength

jseSetAttributes

DESCRIPTION Set the attributes of a jseVariable.

SYNTAX
```
void
jseSetAttributes(jseContext jsecontext,
                 jseVariable variable,
                 jseAttributes attr );
```

COMMENTS

jseContext - The current executing context.

variable - The variable to have its attributes updated.

attr - The attributes to be applied to variable.

The return action can be any of the following values OR'ed together:

jseDefaultAttr - Standard ECMAScript behavior.

jseDontEnum - Ignore this member during for...in enumerations.

jseDontDelete - Cannot be deleted by the delete operator.

jseReadOnly - Makes the variable read only.

jseImplicitThis - Puts the 'this' variable in the scoping chain. This only applies to calling this member if it is in fact a function.

jseImplicitParents - please see the definition of this flag under jseVarAttributes in the Types & Macros chapter for more information.

RETURN None.

JseSetGlobalObject

DESCRIPTION By calling jseSetGlobalObject() and then calling jseInterpret(), you can determine where the functions in that script go. If you interpret several scripts, each with their own global object, the functions will get put into their own space.

SYNTAX
```
Void jseSetGlobalObject(jseContext jsecontext,
                        jseVariable variable)
```

COMMENTS This function may leave temporary variables on the local context, to be cleaned up only when leaving the local context. So if this function is called from the top-level context (i.e., not within a wrapper function), it is more efficient to use:

```
jseSetGlobalObjectEx(jsecontext,
                     variable,
                     jseCreateVar).
```

SEE ALSO Appendix II, Topic 3…, Changing The Global Object

jseTerminateEngine

DESCRIPTION A call to terminate the ScriptEase Interpreter Engine.

SYNTAX
```
void
jseTerminateEngine( void );
```

COMMENTS Call this function after all jseContext links have been terminated. This function cleans up all the resources allocated and initialized by jseInitializeEngine().

RETURN None.

SEE ALSO jseInitializeEngine

jseTerminateExternalLink

DESCRIPTION Terminate a link to a given jseContext.

SYNTAX
```
void
jseTerminateExternalLink(jseContext jsecontext);
```

COMMENTS This routine is used to terminate the given jseContext. After this call, any references to the supplied context are invalid and will cause an error to occur.

jseContext - The ScriptEase context to destroy.

RETURN None.

SEE ALSO jseInitializeExternalLink, jseGetExternalLinkparameters

jseVarNeed

DESCRIPTION Check the type of a given ScriptEase argument variable.

SYNTAX

```
jsebool
jseVarNeed(jseContext jsecontext,
   jseVariable variable,
   jseVarNeeded need );
```

COMMENTS This function verifies that a function argument to a ScriptEase wrapper function is of a given type.

jseContext - The current executing context. Use the value supplied to the wrapper function by the ScriptEase Engine.

variable - The variable being checked for type.

need - The type of the argument you are trying to verify. It can be one of the values specified in jseFuncVarNeed.

RETURN *True* if the variable specified is of the type specified or can be converted according to the flags described in jseFuncVarNeed. *False* otherwise and a error message will have been called.

SEE ALSO jseGetVar, jseFuncVar, jseFuncVarNeed

ScriptEase JavaScript Language

ScriptEase is a scripting or programming language that allows a computer user or programmer to write simple scripts with tremendous power. The guiding principles for ScriptEase are **simplicity** and **power** which add up to easy elegance in scripting. Scripts are much easier to write and use than the source code for compiled languages such as C++.

ScriptEase uses JavaScript, one of the most popular scripting language in today's world, as its core language. In fact, ScriptEase uses the ECMAScript standard for JavaScript. ECMAScript is the core version of JavaScript which has been standardized by the European Computer Manufacturers Association and is the only standardization of JavaScript. ScriptEase closely follows and will follow this standardized JavaScript.

ScriptEase is not limited to JavaScript, as good as it may be. ScriptEase has enhanced the power of JavaScript by adding two objects, Clib and SElib, that have the power of the C programming language. Indeed, ScriptEase implements a scripting version of C which has the power of C in a simple scripting language. With the power of C readily available, computer users or programmers are able to accomplish any tasks that they pursue. Both JavaScript and C script can be intermingled in ScriptEase code, which allows scripters flexibility, power, and simplicity.

The following line is a complete script which could be saved as a script file and run as a program. The program simply displays a message, "A simple one line script," on a computer screen.

```
Screen.writeln("A simple one line script")
```

The following code fragment[1] uses a more structured approach to accomplish the same task. JavaScript and C share similar programming styles, such as the main() function shown in this fragment.

```
function main()
{
   Clib.puts("A simple one line script");
}
```

A ScriptEase script may be written using a very straightforward scripting approach as shown in the first example above, which is similar to the simple scripting of a DOS batch file. A second line could be added to the single line as shown in the following fragment.

```
Screen.writeln("A simple one line script")
Clib.puts("Now there are two lines")
```

[1] "Code fragment" and "fragment" are used interchangeably. They both refer to lines of script or code that perform some scripting or programming task. The lines of code may or may not be complete scripts or programs.

The example using the main() function could be expanded as follows.

```
function main()
{
        Clib.puts("A simple one line script");
        Screen.writeln("Now there are two lines");
}
```

These examples illustrate how easily ScriptEase can be used in a simple scripting mode and how easily the power of functions can be put in a script, and not just the power of functions, but the power of C. They show how easily JavaScript and C script can be intermingled, since C is implemented as a JavaScript object. Functions and other programming concepts are explained in the following descriptions of the ScriptEase language. A tutorial section provides illustrations of scripts in addition to the example code fragments in the text.

Most JavaScript, other than ScriptEase, is part of web browsers and is used while users are connected to the Internet. Usually people are unaware that JavaScript is commonly being executed on their computers when they are connected to various Internet sites. Not only are they unaware, they are unable to write and execute scripts on their computers for their own uses. ScriptEase steps in at this point. Users do not have to be connected to the Internet to use ScriptEase, as they must be with other JavaScript interpreters.

Whether the desire is to write a simple script to copy a document to a backup folder or to write an entire data processing program, ScriptEase can do the job or any other job desired. ScriptEase has joined JavaScript and C. Further, ScriptEase adds commands and functions not available in standard implementations of either. In short, ScriptEase is the most powerful and advanced scripting language available today, and it achieves its power while still being simple to use.

The following sections of this manual will help you to start enjoying the power of ScriptEase.

Basics

Case sensitivity

ScriptEase is case sensitive. A variable named "testvar" is a different variable than one named "TestVar", and both of them can exist in a script at the same time. Thus, the following code fragment defines two separate variables:

```
var testvar = 5
var TestVar = "five"
```

All identifiers in ScriptEase are case sensitive. For example, to display the word "dog" on the screen, the Screen.write() method could be used: Screen.write("dog"). But, if the capitalization is changed to something like, Screen.Write("dog"), then the ScriptEase interpreter generates an error message. Control statements and preprocessor directives are also case sensitive. For example, the statement "while" is valid, but the word "While" is not. The directive "#if" works, but the letters "#IF" fail.

Whitespace characters

Whitespace characters, space, tab, carriage-return and new-line, govern the spacing and placement of text. Whitespace makes code more readable for humans, but is ignored by the interpreter2.

Lines of script end with a carriage-return, and each line is usually a separate statement. (Technically, in many editors, lines end with a carriage-return and linefeed pair, "\r\n".) Since the interpreter usually sees one or more whitespace characters between identifiers as simply whitespace, the following ScriptEase statements are equivalent to each other:

```
var x=a+b
var x = a + b
var x =          a          +          b
var x = a
       + b
```

Whitespace separates identifiers into separate entities. For example, "ab" is one variable name, and "a b" is two. Thus, the fragment, "`var a b = 2`" is valid, but "`var ab = 2`" is not.

Many programmers use all spaces and no tabs, because tab size settings vary from editor to editor and programmer to programmer. By using spaces only, the format of a script will look the same on all editors. All scripts provided by Nombas with ScriptEase use spaces only.

Comments

A comment is text in a script to be read by humans and not the interpreter which skips over comments. Comments help people to understand the purpose and program flow of a program. Good comments, which explain lines of code well, help people alter code that they have written in the past or that was written by someone else.

There are two formats for comments: end of line comments and block comments. End of line comments begin with two slash characters, "//". Any text after two consecutive slash characters is ignored to the end of the current line. The interpreter begins interpreting text as code on the next line. Block comments are enclosed within a beginning block comment, "/*", and an end of block comment, "*/". Any text between these markers is a comment, even if the comment extends over multiple lines. Block comments may not be nested within block comments, but end of line comments can exist within block comments.

---

2 The phrase, "the interpreter," is used synonymously with, "the ScriptEase interpreter." ScriptEase, like JavaScript and many other popular languages, is an interpreted language.

The following code fragments are examples of valid comments:

```
// this is an end of line comment

/* this is a block comment
 This is one big comment block.
 // this comment is okay inside the block
 Isn't it pretty?
*/

var FavoriteAnimal = "dog"; // except for poodles

//This line is a comment but
var TestStr = "this line is not a comment";
```

Expressions, statements, and blocks

An expression or statement is any sequence of code that performs a computation or an action, such as the code "`var TestSum = 4 + 3`" which computes a sum and assigns it to a variable. ScriptEase code is executed one statement at a time in the order in which it is read. Many programmers put semicolons at the end of statements, although they are not required. Each statement is usually written on a separate line, with or without semicolons, to make scripts easier to read and edit.

A statement block is a group of statements enclosed in curly braces, "{}", which indicate that the enclosed individual statements are a group and are to be treated as one statement. A block can be used anywhere that a single statement can.

A *while* statement causes the statement after it to be executed in a loop. By enclosing multiple statements in curly braces, they are treated as one statement and are executed in the while loop. The following fragment illustrates:

```
while( ThereAreUncalledNamesOnTheList() == True)
{
          var name = GetNameFromTheList();
          CallthePerson(name);
          LeaveTheMessage();
}
```

All three lines after the while statement are treated as a unit. If the braces were omitted, the *while* loop would only apply to the first line. With the braces, the script goes through all lines until everyone on the list has been called. Without the braces, the script goes through all names on the list, but only the last one is called. Two very different procedures.

Statements within blocks are often indented for easier reading.

Identifiers

Identifiers are merely names for variables and functions. Programmers must know the names of built in variables and functions to use them in scripts and must know some rules about identifiers to define their own variables and functions. The following rules are simple and intuitive.

- Identifiers may use only ASCII letters, upper or lower case, digits, the underscore, "_", and the dollar sign, "$". That is, they may use only characters from the following sets of characters.

  ```
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  "abcdefghijklmnopqrstuvwxyz"
  "0123456789"
  "_$"
  ```

- Identifiers may **not** use letters of the following characters.

  ```
  "+-<>&|=!*/%^~?:{};()[].'"'#,"
  ```

- Identifiers must begin with a letter, underscore, or dollar sign, but may have digits anywhere else.
- Identifiers may not have whitespace in them since whitespace separates identifiers for the interpreter.
- Identifiers may be as long a programmer needs.

The following identifiers, variables and functions, are valid:

```
Sid
Nancy7436
annualReport
sid_and_nancy_prepared_the_annualReport
$alice
CalculateTotal()
$SubtractLess()
Divide$All()
```

The following identifiers, variables and functions, are not valid:

```
1sid
2nancy
this&that
Sid and Nancy
ratsAndCats?
=Total()
(Minus)()
Add Both Figures()
```

Prohibited identifiers

The following words have special meaning for the interpreter and cannot be used as identifiers, neither as variable nor function names:

break	case	catch	class	const	continue	debugger
default	delete	do	else	enum	export	extends
False	finally	for	function	if	import	in
new	NULL	return	super	switch	this	throw
True	try	typeof	while	with	var	void

Variables

A variable is an identifier to which data may be assigned. Variables are used to store and represent information in a script. Variables may change their values, but literals may not. For example, if programmers want to display a name literally, they must use something like the following fragment multiple times.

```
Screen.writeln("Rumpelstiltskin Henry Constantinople")
```

But they could use a variable to make their task easier, as in the following.

```
var Name = "Rumpelstiltskin Henry Constantinople"
Screen.write(Name)
```

Then they can use shorter lines of code for display and use the same lines of code repeatedly by simply changing the contents of the variable Name.

Variable scope

Variables in ScriptEase may be either global or local. Global variables may be accessed and modified from anywhere in a script. Local variables may only be accessed from the functions in which they are created. There are no absolute rules for preferring or using global or local variables. Each type has merit. In general, programmers prefer to use local variables when reasonable since they facilitate modular code that is easier to alter and develop over time. It is generally easier to understand how local variables are used in a single function than how global variables are used throughout an entire program. Further, local variables conserve system resources.

To make a local variable, declare it in a function using the var keyword:

```
var perfectNumber;
```

A value may be assigned to a variable when it is declared:

var perfectNumber = 28;

The default behavior of ScriptEase is that variables declared outside of any function or inside a function without the *var* keyword are global variables. However, this behavior can be changed by the DefaultLocalVariables and RequireVarKeyword settings of the #option preprocessor directive. This directive is explained in the section on preprocessing. For now, consider the following code fragment.

```
var a = 1;
function main()
{
         b = 1;
         var d = 3;
         someFunction(d);
}

function someFunction(e)
{
         var c = 2
         ...
}
```

In this example, *a* and *b* are both global variables, since *a* is declared outside of a function and *b* is defined without the *var* keyword. The variables, *d* and *c*, are both local, since they are defined within functions with the *var* keyword. The variable *c* may not be used in the *main*() function, since it is undefined in the scope of that function. The variable *d* may be used in the *main*() function and is explicitly passed as an argument to *someFunction*() as the parameter *e*. The following lines show which variables are available to the two functions:

```
main():      a, b, d
someFunction():     a, b, c, e
```

It is possible, though not usually a good idea, to have local and global variables with the same name. In such a case, a global variable must be referenced as a property of the global object, and the variable name used by itself refers to the local variable. In the fragment above, the global variable *a* can be referenced anywhere in its script by using: "global.a".

Functions

Functions are identified by names, as variables are. Functions perform script operations, and variables store data. Functions do the work of a script and will be discussed in more detail later. The reason they are mentioned here is simply to point out that they have identifiers, names, that follow the same rules for identifiers as variable names do.

Function scope

Functions are all global in scope, much like global variables. A function may not be declared within another function so that its scope is merely within a certain function or section of a script. All functions may be called from anywhere in a script. If it is helpful, think of functions as methods of the global object. The following two code fragments do exactly the same thing. The first calls a function, SumTwo(), as a function, and the second calls SumTwo() as a method of the global object.

```
// fragment one
function SumTwo(a, b)
{
```

```
        return a + b
}

Screen.writeln(SumTwo(3, 4))


// fragment two
function SumTwo(a, b)
{
        return a + b
}

Screen.writeln(global.SumTwo(3, 4))
```

Data types

Data types in ScriptEase can be classified into three groupings: primitive, composite, and special. In a script, data can be represented by literals or variables. The following lines illustrates variables and literals:

```
var TestVar = 14;
var aString = "test string";
```

The variable *TestVar* is assigned the literal 14, and the variable *aString* is assigned the literal "test string". After these assignments of literal values to variables, the variables can be used anywhere in a script where the literal values could to be used.

In the fragment above which defines and uses the function SumTwo(), the literals, 3 and 4, are passed as arguments to the function SumTwo() which has corresponding parameters, a and b. The parameters, a and b, are variables for the function that hold the literal values that were passed to it.

Data types need to be understood in terms of their literal representations in a script and of their characteristics as variables.

Data , in literal or variable form, is assigned to a variable with an assignment operator which is often merely an equal sign, "=" as the following lines illustrate.

```
var happyVariable = 7;
var joyfulVariable = "free chocolate";
var theWorldIsFlat = True;
var happyToo = happyVariable;
```

The first time a variable is used, its type is determined by the interpreter, and the type remains until a later assignment changes the type automatically. The example above creates three variables, each of a different type. The first is a number, the second is a string, and the third is a boolean variable. Variable types are described below. Since ScriptEase automatically converts variables from one type to another when needed, programmers normally do not have to worry about type conversions as they do in strongly typed languages, such as C.

Primitive data types

Variables that have primitive data types pass their data by value, by actually copying the data to the new location. The following fragment illustrates:

```
var a = "abc";
var b = ReturnValue(a);

function ReturnValue(c)
{
         return c;
}
```

After "abc" is assigned to variable a, two copies of the string "abc" exist, the original literal and the copy in the variable a. While the function ReturnValue is active, the parameter/variable c has a copy, and three copies of the string "abc" exist. If c were to be changed in such a function, variable a, which was passed as an argument to the function, would remain unchanged. After the function ReturnValue is finished, a copy of "abc" is in the variable b, but the copy in the variable c in the function is gone because the function is finished. During the execution of the fragment, as many as three copies of "abc" exist at one time.

The primitive data types are: Number, Boolean, and String.

Number

Integer

Integers are whole numbers. Decimal integers, such as 1 or 10, are the most common numbers encountered in daily life. ScriptEase has three notations for integers: decimal, hexadecimal, and octal.

Decimal

Decimal notation is the way people write numbers in everyday life and uses base 10 digits from the set of 0-9. Examples are:

```
1, 10, 0, and 999
var a = 101;
```

Hexadecimal

Hexadecimal notation uses base 16 digits from the sets of 0-9, A-F, and a-f. These digits are preceded by 0x. ScriptEase is not case sensitive when it comes to hexadecimal numbers. Examples are:

```
0x1, 0x01, 0x100, 0x1F, 0x1f, 0xABCD
var a = 0x1b2E;
```

Octal

Octal notation uses base 8 digits from the set of 0-7. These digits are preceded by 0. Examples are:

```
00, 05, and 077
var a = 0143;
```

Floating point

Floating point numbers are numbers with fractional parts which are often indicated by a period, for example, 10.33. Floating point numbers are often referred to as floats.

Decimal

Decimal floats use the same digits as decimal integers but allow a period to indicate a fractional part. Examples are:

```
0.32, 1.44, and 99.44
var a = 100.55 + .45;
```

Scientific

Scientific floats are often used in the scientific community for very large or small numbers. They use the same digits as decimals plus exponential notation. Scientific notation is sometimes referred to as exponential notation. Examples are:

```
4.087e2, 4.087E2, 4.087e+2, and 4.087E-2
var a = 5.321e33 + 9.333e-2;
```

Boolean

Booleans may have only one of two possible values: *false* or *true*. Since ScriptEase automatically converts values when appropriate, Booleans can be used as they are in languages such as C. Namely, *false* is zero, and *true* is non-zero. A script is more precise when it uses the actual ScriptEase values, *false* and *true*, but it will work using the concepts of zero and not zero. When a Boolean is used in a numeric context, it is converted to 0, if it is *false*, and 1, if it is *true*.

String

A String is a series of characters linked together. A string is written using quotation marks, for example: "I am a string", 'so am I', 'me too', and "344". The string "344" is different from the number 344. The first is an array of characters, and the second is a value that may be used in numerical calculations.

ScriptEase automatically converts strings to numbers and numbers to string, depending on context. If a number is used in a string context, it is converted to a string. If a string is used in a number context, it is converted to a numeric value. Automatic type conversion is discussed more fully in a later section

Strings, though classified as a primitive, are actually a hybrid type that shares characteristics of primitive and composite data types. Strings are discussed more fully a later section.

Composite data types

Whereas primitive types are passed by value, composite types are passed by reference. When a composite type is assigned to a variable or passed to a parameter, only a reference that points to its data is passed.

The following fragment illustrates.

```
var AnObj = new Object;
AnObj.name = "Joe";
AnObj.old = ReturnName(AnObj)

function ReturnName(CurObj)
{
          return CurObj.name
}
```

After the object *AnObj* is created, the string "Joe" is assigned, by value since a property is a variable within an Object, to the property *AnObj.name*. Two copies of the string "Joe" exist. When *AnObj* is passed to the function *ReturnName*, it is passed by reference. *CurObj* does not receive a copy of the Object, but only a reference to the Object. With this reference, *CurObj* can access every property and method of the original. If *CurObj.name* were to be changed while the function was executing, then *AnObj.name* would be changed at the same time. When *AnObj.old* receives the return from the function, the return is assigned by value, and a copy of the string "Joe" transferred to the property. Thus, *AnObj* holds two copies of the string "Joe": one in the property *.name* and one in the property *.old*. Three total copies of "Joe" exist, counting the original string literal.

Two commonly used composite data types are: Object and Array.

Object

An object is a compound data type, consisting of one or more pieces of data of any type which are grouped together in an object. Data that are part of an object are called properties of the object. The Object data type is similar to the structure data type in C and in previous versions of ScriptEase. The object data type also allows functions, called methods, to be used as object properties. Indeed, in ScriptEase, functions are considered to be like variables. But for practical programming, think of objects as having methods, which are functions, and properties, which are variables and constants.

Objects and their characteristics are discussed more fully in a later section.

Array

An array is a series of data stored in a variable that is accessed using index numbers that indicate particular data. The following fragments illustrate the storage of the data in separate variables or in one array variable:

```
var Test0 = "one";
var Test1 = "two";
var Test2 = "three";

var Test = new Array;
Test[0] = "one";
Test[1] = "two";
Test[2] = "three";
```

After either fragment is executed, the three strings are stored for later use. In the first fragment, three separate variables have the three separate strings. These variables must be used separately. In the second fragment, one variable holds all three strings. This Array variable can be used as one unit, and the strings can be accessed individually. The similarities, in grouping, between Arrays and Objects is more than slight. In fact, Arrays and Objects are both objects in ScriptEase with different notations for accessing properties. For practical programming, Arrays may be considered as a data type of their own.

Arrays and their characteristics are discussed more fully in a later section.

Special values

undefined

If a variable is created or accessed with nothing assigned to it, it is of type undefined. An undefined variable merely occupies space until a value is assigned to it. When a variable is assigned a value, it is assigned a type according to the value assigned. Though variables may be of type undefined, there is no literal representation for undefined. Consider the following invalid fragment.

```
var test;
if (typeof test == "undefined")
          Screen.writeln("test is undefined")
```

After var test is declared, it is undefined since no value has been assigned to it. But, the test, "`test == undefined`", is invalid because there is no way to literally represent undefined.

NULL

NULL is a special data type that indicates that a variable is empty, a condition that is different from being undefined. A null variable holds no value, though it might have previously. The null type is represented literally by the identifier, null. Since ScriptEase automatically converts data types, null is both useful and versatile. The code fragment above will work if "undefined" is changed to "null", as shown in the following:

```
var test = null;
if( test==null )
       Screen.writeln("It is null.");
```

Since null has a literal representation, assignments like the following are valid:

```
var test = null;
```

Any variable that has been assigned a value of null can be compared to the null literal.

NaN

The NaN type means "Not a Number". NaN is merely an acronym for the phrase. However, NaN does not have a literal representation. To test for NaN, the function, isNaN(), must be used, as illustrated in the following fragment:

```
var Test = "a string";
if (isNaN(parseInt(Test)))
          Screen.writeln("Test is Not a Number");
```

When the parseInt() function tries to parse the string "a string" into an integer, it returns NaN, since "a string" does not represent a number like the string "22" does.

Number constants

Several numeric constants can be accessed as properties of the Number object, though they do not have a literal representation.

Constant	Value	Description
Number.MAX_VALUE	1.7976931348623157e+308	Largest number (positive)
Number.MIN_VALUE	2.2250738585072014e-308	Smallest positive non-zero value
Number.NaN	NaN	Not a Number
Number.POSITIVE_INFINITY	Infinity	Number above MAX_VALUE
Number.NEGATIVE_INFINITY	Infinity	Number below MIN_VALUE

Automatic type conversion

When a variable is used in a context where it makes sense to convert it to a different type, ScriptEase automatically converts the variable to the appropriate type. Such conversions most commonly happen with numbers and strings. For example:

```
"dog" + "house" == "doghouse" // two strings are joined
"dog" + 4 ==  "dog4"        // a number is converted
4 + "4" == "44"     // to a string
4 + 4 == 8   // two numbers are added
23 - "17" == 6      // a string is converted to a number
```

Converting numbers to strings is fairly straightforward. However, when converting strings to numbers there are several limitations. While subtracting a string from a number or a number from a string converts the string to a number and subtracts the two, adding the two converts the number to a string and concatenates them. String always convert to a base 10 number and must not contain any characters other than digits. The string "110n" will not convert to a number, because the ScriptEase interpreter does not know what to make of the "n" character.

You can specify more stringent conversions by using the global methods, *parseInt()* and *parseFloat()* methods. Further, ScriptEase has many global functions to cast data as a specific type, functions that are not part of the ECMAScript standard. These functions are described in the section on global functions that are specific to ScriptEase.

Properties and methods of basic data types

The basic data types, such as Number and String, have properties and methods assigned to them that may be used with any variable of that type. For example, all String variables may use all String methods.

The properties and methods of the basic data types are retrieved in the same way as from objects. For the most part, they are used internally by the interpreter, but you may use them if choose. For example, if you have a numeric variable called number and you want to convert it to a string, you can use the .toString() method as illustrated in the following fragment.

```
var n = 5
var s = n.toString()
```

After this fragment executes, the variable n contains the number 5 and the variable s contains the string "5".

The following two methods are common to all variables.

.toString()

This method returns the value of a variable expressed as a string.

.valueOf()

This method returns the value of a variable.

Operators

Mathematical operators

Mathematical operators are used to make calculations using mathematical data. The following sections illustrate the mathematical operators in ScriptEase.

Basic arithmetic

The arithmetic operators in ScriptEase are pretty standard.

```
=      assignment          assigns a value to a variable
+      addition            adds two numbers
-      subtraction         subtracts a number from another
*      multiplication      multiplies two numbers
/      division            divides a number by another
%      modulo returns a remainder after division
```

The following are examples using variables and arithmetic operators.

```
var i;
i = 2;       i is now  2
```

```
i = i + 3;    i is now  5, (2+3)
i = i - 3;    i is now  2, (5-3)
i = i * 5;    i is now 10, (2*5)
i = i / 3;    i is now  3, (10/3) (the remainder is ignored)
i = 10;       i is now 10
i = i % 3;    i is now  1, (10%3)
```

Expressions may be grouped to affect the sequence of processing. All multiplication and division is calculated for an expression before addition and subtraction unless parentheses are used to override the normal order. Expressions inside parentheses are processed first, before other calculations. In the following examples, the information inside square brackets, "[]," are summaries of calculations provided with these examples and not part of the calculations.

Notice that:

```
4 * 7 - 5 * 3; [28 - 15 = 13]
```

has the same meaning, due to the order of precedence, as:

```
(4 * 7) - (5 * 3); [28 - 15 = 13]
```

but has a different meaning than:

```
4 * (7 - 5) * 3; [4 * 2 * 3 = 24]
```

which is still different from:

```
4 * (7 - (5 * 3)); [4 * -8 = -32]
```

The use of parentheses is recommended in all cases where there may be confusion about how the expression is to be evaluated, even when they are not necessary.

Assignment arithmetic

Each of the above operators can be combined with the assignment operator, =, as a shortcut for performing operations. Such assignments use the value to the right of the assignment operator to perform an operation with the value to the left. The result of the operation is then assigned to the value on the left.

```
 =     assignment    assigns a value to a variable
+=     assign addition     adds a value to a variable
-=     assign subtraction  subtracts a value from a variable
       *=     assign multiplication     multiplies a variable by a
              value
/=     assign division     divides a variable by a value
       %=     assign remainder    returns a remainder after
              division
```

The following lines are examples using assignment arithmetic.

```
var i;
i  = 2;       i is now  2
i += 3;       i is now  5, (2+3)         same as i = i + 3
i -= 3;       i is now  2, (5-3)         same as i = i - 3
i *= 5;       i is now 10, (2*5)         same as i = i * 5
i /= 3;       i is now  3.333, (10/3)    same as i = i / 3
i  = 10;      i is now 10
i %= 3;       i is now  1, (10%3)        same as i = i % 3
```

Auto-increment (++) and auto-decrement (--)

To add or subtract one, 1, to or from a variable, use the auto-increment, ++, or auto-decrement, --, operator. These operators add or subtract 1 from the value to which they are applied. Thus, "`i++`" is a shortcut for "`i += 1`", which is a shortcut for "`i = i + 1`".

These operators can be used before, as a prefix operator, or after, as a postfix operator, their variables. If they are used before a variable, it is altered before it is used in a statement, and if used after, the variable is altered after it is used in the statement. The following lines demonstrates prefix and postfix operations.

```
i = 4; i is 4
     j = ++i;    j is 5, i is 5    (i was incremented before use)
     j = i++;    j is 5, i is 6    (i was incremented after use)
     j = --i;    j is 5, i is 5    (i was decremented before use)
     j = i--;    j is 5, i is 4    (i was decremented after use)
     i++;        i is 5            (i was incremented)
```

Bit operators

ScriptEase contains many operators for operating directly on the bits in a byte or an integer. Bit operations require a knowledge of bits, bytes, integers, binary numbers, and hexadecimal numbers. Not every programmer needs to or will choose to use bit operators.

```
<<      shift left                i = i << 2;
<<=     assignment shift left     i <<= 2;
>>      shift right               i = i >> 2;
>>=     assignment shift right    i >>= 2;
>>>     shift left with zeros     i = i >>> 2
>>>=    assignment shift left     i >>>= 2
        with zeros
&       bitwise and               i = i & 1
&=      assignment bitwise and    i &= 1;
|       bitwise or                i = i | 1
|=      assignment bitwise or     i |= 1;
^       bitwise xor, exclusive    i = i ^ 1
        or
^=      assignment bitwise xor,   i ^= 1
        exclusive or
~       Bitwise not, complement   i = ~i;
```

Logical operators and conditional expressions

Logical operators compare two values and evaluate whether the resulting expression is *false* or *true*. A variable or any other expression may be *false* or *true*. An expression that does a comparison is called a conditional expression.

Logical operators are used to make decisions about which statements in a script will be

executed, based on how a conditional expression evaluates. As an example, suppose that you are designing a simple guessing game. The computer thinks of a number between 1 and 100, and you guess what it is. The computer tells you if you are right or not and whether your guess is higher or lower than the target number. This procedure uses the *if* statement, which is introduced in the next section. Basically, if the conditional expression in the parenthesis following an if statement is *true*, the statement block following the *if* statement is executed. If *false*, the statement block is ignored, and the computer continues executing the script at the next statement after the ignored block.

The script might have a structure similar to the one following, in which *GetTheGuess()* is a function that gets your guess.

```
var guess = GetTheGuess(); //get the user input
if (guess > target_number)
{
          guess is too high...
}

if (guess < target_number)
{
          guess is too low...
}

if (guess == target_number)
{
          you guessed the number!...
}
```

This example is simple, but it illustrates how logical operators can be used to make decisions in ScriptEase.

The logical operators are:

`!`	not	reverses an expression. If `(a+b)` is *true*, then `!(a+b)` is *false*.
`&&`	and	*true* if, and only if, both expressions are *true*. Since both expressions must be *true* for the statement as a whole to be *true*, if the first expression is *false*, there is no need to evaluate the second expression, since the whole expression is *false*.
`||`	or	*true* if either expression is *true*. Since only one of the expressions in the or statement needs to be *true* for the expression to evaluate as *true*, if the first expression evaluates as *true*, the interpreter returns *true* and does not bother with evaluating the second.
`==`	equality	*true* if the values are equal, otherwise *false*. Do not confuse the equality operator, ==, with the assignment operator, =.
`!=`	inequality	*true* if the values are not equal, else *false*.
`<`	less than	a < b is *true* if a is less than b.
`>`	greater than	a > b is *true* if a is greater than b.
`<=`	less than or equal to	a <= b is *true* if a is less than or equal to b.
`>=`	greater than or equal to	a >= b is *true* if a is greater than b.

Remember, the assignment operator, =, is different than the equality operator, ==. If you use one equal sign when you intend two, your script will not function the way you want it to. This is a common pitfall, even among experienced programmers. The two meanings of equal signs must be kept separate, since there are times when you have to use them both in the same statement, and there is no way the computer can differentiate them by context.

typeof operator

The typeof operator provides a way to determine and to test the data type of a variable and may use either of the following notations, with or without parentheses.

```
var result = typeof variable
var result = typeof(variable)
```

After either line, the variable result is set to a string that is represents the variable's type: "undefined", "boolean", "string", "object", "number", "function" or "buffer".

Flow decisions statements

This section describes statements that control the flow of a program. Use these statements to make decisions and to repeatedly execute statement blocks.

if

The *if* statement is the most commonly used mechanism for making decisions in a program. It allows you to test a condition and act on it. If an *if* statement finds the condition you test to be *true*, the statement or statement block following it are executed. The following fragment is an example of an if statement.

```
if ( goo < 10 )
{
          Screen.write("goo is smaller than 10\n");
}
```

else

The *else* statement is an extension of the *if* statement. It allows you to tell your program to do something else if the condition in the *if* statement was found to be *false*. In ScriptEase code, it looks like the following.

```
if ( goo < 10 )
{
          Screen.write("goo is smaller than 10\n");
}
else
{
          Screen.write("goo is not smaller than 10\n");
}
```

To make more complex decisions, *else* can be combined with if to match one out of a number of possible conditions.

The following fragment illustrates using *else* with *if*.

```
if ( goo < 10 )
{
          Screen.write("goo is less than 10\n");
          if ( goo < 0 )
          {
Screen.write("goo is negative; so it's less than 10\n");
          }
}
else if ( goo > 10 )
{
          Screen.write("goo is greater than 10\n");
}
else
{
          Screen.write("goo is 10\n");
}
```

while

The *while* statement is used to execute a particular section of code, over and over again, until an expression evaluates as *false*.

```
while (expression)
{
          DoSomething();
}
```

When the interpreter comes across a while statement, it first tests to see whether the expression is *true* or not. If the expression is *true*, the interpreter carries out the statement or statement block following it. Then the interpreter tests the expression again. A while loop repeats until the test expression evaluates to *false*, whereupon the program continues after the code associated with the *while* statement.

The following fragment illustrates a *while* statement with a two lines of code in a statement block.

```
while( ThereAreUncalledNamesOnTheList() != false)
{
          var name=GetNameFromTheList();
          SendEmail(name);
}
```

do {...} while

The *do* statement is different from the *while* statement in that the code block is executed at least once, before the test condition is checked.

```
var value = 0;
do
{
          value++;
          ProcessData(value);
} while( value < 100 );
```

The code used to demonstrate the *while* statement could also be written as the following fragment.

```
do
{
          var name = GetNameFromTheList();
          SendEmail(name)
} while (name != TheLastNameOnTheList());
```

Of course, if there are no names on the list, the script will run into problems!

for

The *for* statement is a special looping statement. It allows for more precise control of the number of times a section of code is executed. The *for* statement has the following form.

```
for ( initialization; conditional; loop_expression )
{
          statement
}
```

The *initialization* is performed first, and then the expression is evaluated. If the result is *true* or if there is no *conditional* expression, the statement is executed. Then the *loop_expression* is executed, and the *expression* is re-evaluated, beginning the loop again. If the *expression* evaluates as *false*, then the *statement* is not executed, and the program continues with the next line of code after the *statement*. For example, the following code displays the numbers from 1 to 10.

```
for(var x=1; x<11; x++)
{
          Screen.write(x);
}
```

None of the statements that appear in the parentheses following the for statement are mandatory, so the above code demonstrating the while statement would be rewritten this way if you preferred to use a for statement:

```
 for( ; ThereAreUncalledNamesOnTheList() ; )
 {
            var name=GetNameFromTheList();
            SendEmail(name)
 }
```

Since we are not keeping track of the number of iterations in the loop, there is no need to have an initialization or loop_expression statement. You can use an empty for statement to create an endless loop:

```
for(;;)
{
//the code in this  block will repeat forever,
//unless the program breaks out of the for loop somehow.
}
```

break

Break and *continue* are used to control the behavior of the looping statements: *for*, *while*, and *do*. The *break* statement terminates the innermost loop of *for*, *while*, or *do* statements. The program resumes execution on the next line following the loop. The following code fragment does nothing but illustrate the break statement.

```
for(;;)
{
       break;
}
```

The break statement is also used at the close of a *case* statement, as shown below.

continue

The *continue* statement ends the current iteration of a loop and begins the next. Any conditional expressions are reevaluated before the loop reiterates.

switch, case, and default

The *switch* statement makes a decision based on the value of a variable or statement. The *switch* statement follows the following format:

```
switch( switch_variable )
{
case value1:
       statement1
       break;
case value2:
       statement2
       break;
       .
       .
       .
default:
       default_statement
}
```

The variable *switch_variable* is evaluated, and then it is compared to all of the values in the *case* statements (*value1*, *value2*, . . . , *default*) until a match is found. The statement or statements following the matched case are executed until the end of the *switch* block is reached or until a *break* statement exits the *switch* block. If no match is found, the *default* statement is executed, if there is one.

For example, suppose you had a series of account numbers, each beginning with a letter that determines what type of account it is. You could use a switch statement to carry out actions depending on that first letter. The same task could be accomplished with a series of nested *if* statements, but they require much more typing and are harder to read.

```
switch ( key[0] )
{
case 'A':
       Screen.write("A"); //handle 'A' accounts...
       break;
case 'B':
       Screen.write("B"); //handle 'B' accounts...
       break;
case 'C':
       Screen.write("C"); //handle 'C' accounts...
       break;
       default:
       Screen.write("Invalid account number.\n");
       break;
}
```

A common mistake is to omit a *break* statement to end each case. In the preceding example, if the *break* statement after the Screen.write("B") statement were omitted, the computer would print both "B" and "C", since the interpreter executes commands until a *break* statement is encountered.

goto and labels

You may jump to any location within a function block by using the *goto* statement. The syntax is:

```
goto LABEL;
```

where *LABEL* is an identifier followed by a colon (:). The following code fragment continuously prompts for a number until a number less than 2 is entered.

```
beginning:
Screen.write("Enter a number less than 2:")
var x = getche();     //get a value for x
if (a >= 2)
       goto beginning;
Screen.write(a);
```

As a rule, *goto* statements should be used sparingly, since they make it difficult to track program flow.

Conditional operator ? :

The conditional operator provides a shorthand method for writing else statements. It is harder to read than conventional *if* statements, and so is generally used when the expressions in the *if* statements are brief. The syntax is:

```
test_expression ? expression_if_true : expression_if_false
```

First, *test_expression* is evaluated. If *test_expression* is *true*, then *expression_if_true* is evaluated, and the value of the entire expression replaced by the value of *expression_if_true*. If *test_expression* is *false*, then *expression_if_false* is evaluated, and the value of the entire expression is that of *expression_if_false*.

The following fragment illustrates the use of the conditional operator.

```
foo = ( 5 < 6 ) ? 100 : 200; // foo is set to 100
Screen.write("Name is " + ((null==name) ? "unknown" : name));
```

Functions

A function is an independent section of code that receives information from a program and performs some action with it. Once a function has been written, you do not have to think again about how to perform the operations in it. Just call the function, and let it handle the work for you. You only need to know what information the function needs to receive, that is, the parameters, and whether it returns a value to the statement that called it.

Screen.write() is an example of a function which provides an easy way to display formatted text. It receives a string from the function that called it and displays the string on the screen. Screen.write is a void function, meaning it has no return value.

In JavaScript, functions are considered a data type, evaluating to whatever the function's return value is. You can use a function anywhere you can use a variable. Any valid variable name may be used as a function name. Like comments, using descriptive function names helps you keep track of what is going on with your script.

Two rules set functions apart from the other variable types: instead of being declared with the "var" keyword, functions are declared with the "function" keyword, and functions have the function operator, "()", following their names. Data to be passed to a function is included within these parentheses.

Several sets of built-in functions are included as part of the ScriptEase interpreter. These functions are described in this manual. They are internal to the interpreter and may be used at any time. In addition, ScriptEase ships with a number of external libraries or .jsh files. External libraries must be explicitly included in your script to use the functions in them. See the description of the *#include* preprocessor directive.

ScriptEase allows you to have two functions with the same name. The interpreter uses the function nearest the end of the script, that is, the last function to load is the one to be executed when the function name is called. By taking advantage of this behavior, you can write functions that supersede the ones included in the interpreter or .jsh files.

Function return statement

The *return* statement passes a value back to the function that called it. Any code in a function following the execution of a return statement is not executed.

```
function DoubleAndDivideBy5(a)
{
	return (a*2)/5
}
```

Here is an example of a script using the above function.

```
function main()
{
       var a = DoubleAndDivideBy5(10);
       var b = DoubleAndDivideBy5(20);
       Screen.write(a + b);
}
```

This script displays12.

Passing variables to functions

JavaScript uses different methods to pass variables to functions, depending on the type of variable being passed. Such distinctions ensure that information gets to functions in the most complete and logical ways.

Primitive types, namely, Strings, numbers, and Booleans, are passed by value. The value of theses variables are passed to a function. If a function changes one of these variables, the changes will not be visible outside of the function where the change took place.

Composite types, Objects and Arrays, are passed by reference. Instead of passing the value of the object, that is, the values of each property, a reference to the object is passed. The reference indicates where in a computer's memory that values of an object's properties are stored. If you make a change in a property of an object passed by reference, that change will be reflected throughout in the calling routine.

Function properties -- arguments[]

The *arguments*[] property is an array of all of the arguments passed to a function. The first variable passed to a function is referred to as *arguments*[0], the second as *arguments*[1], and so forth.

The most useful aspect of this property is that it allows you to have functions with an indefinite number of parameters. Here is an example of a function that takes a variable number of arguments and returns the sum of them all.

```
function SumAll()
{
         var total = 0;
         for (var ssk = 0; ssk < SumAll.arguments.length; ssk++)
         {
            total += SumAll.arguments[ssk];
         }
         return total;
}
```

Function recursion

A recursive function is a function that calls itself or that calls another function that calls the first function. Recursion is permitted in ScriptEase. Each call to a function is independent of any other call to that function. (See the section on variable scope.) Be aware that recursion has limits. If a function calls itself too many times, a script will run out of memory and abort.

Do not worry if recursion is confusing, since you rarely have to use it. Just remember that a function can call itself if it needs to. For example, the following function, factor(), factors a number. Factoring is an ideal candidate for recursion because it is a repetitive process where the result of one factor is then itself factored according to the same rules.

```
function factor(i) //recursive function to print all factors of i,
{// and return the number of factors in i
         if ( 2 <= i )
         {
            for ( var test = 2; test <= i; test++ )
            {
               if ( 0 == (i % test) )
               {
                 // found a factor, so print this factor then call
                 // factor() recursively to find the next factor
                 return( 1 + factor(i/test) );
               }
            }
         }
// if this point was reached, then factor not found
return( 0 );
}
```

Error checking for functions

Some functions return a special value if they fail to do what they are supposed to do. For example, the Clib.fopen() method opens or creates a file for a script to read from or write to. But suppose that the computer is unable to open a file. In such a case, the Clib.fopen() method returns null.

If you try to read from or write to a file that was not properly opened, you get all kinds of errors. To prevent these errors, make sure that Clib.fopen() does not return null when it tries to open a file. Instead of just calling Clib.fopen() as follows:

```
var fp = Clib.fopen("myfile.txt", "r");
```

check to make sure that null is not returned:

```
if (null == (var fp = Clib.fopen("myfile.txt", "r")))
{
       ErrorMsg("Clib.fopen returned null");
}
```

You may abort a script in such a case, but at least you will know why. See the section on the Clib object.

The main() function

If a script has a function called *main*(), it is the first function executed. (For more information on what takes place when a script is run, see the section on running a script.) Other than the fact that *main*() is the first function executed, it is like other functions. If the *main*() function returns a value, that value is returned to the operating system or whatever process called the script.

The *main*() function automatically receives two parameters, which, by convention, are called *argc* and *argv*. The parameter *argc*, argument count, is the number of parameters passed to the script and the parameter *argv* is an array of strings, with each element being one of the parameters. The first element, *argv*[0], of this array is always the name of the script, thus if *argc* == 1, then no variables were passed to a script.

Arguments are passed to a script as parameters when it is called from a command line as illustrated in the following line.

```
sewin32.exe jseedit.jse document.txt
```

In the example above, *argc* == 2, *argv*[0] == "jseedit.jse" and *argv*[1] == "document.txt".

The cfunction keyword

The cfunction keyword defines a function whose behavior is somewhat different than that of standard functions. In a cfunction, variables and operators behave more as they would in C, specifically in the ScriptEase implementation of C as a scripting language. The cfunction is provided for the convenience of C programmers who are used to the way the C language handles functions and variables and for those situations in which the underlying logic of C is more efficient for a particular procedure.

You can change the contents of strings or parts of them by assigning a new character value to a element of a character array. For example:

```
var string = "file"
string[0] = 'm'
```

This fragment creates a string containing the word "mile".

Array arithmetic

If you try to add a number to a string, instead of converting the number to a string and concatenating the two, the starting point of the string will be shifted forward by the number of characters in number.

For example, the statement:

```
"This is a test" + 3
```

evaluates to "This is a test3", in a standard JavaScript. In a *cfunction*, however, this statement evaluates to "s is a test". The starting point of the string has been shifted by three, so that *string*[0] is now 's' instead of 'T'. The 'T', 'h', and 'i' of the original string are at indices [-3], [-2], and [-1], respectively.

Variables are passed to cfunctions by reference. In other words, if you have two variables:

```
var George = "one"
var Martha = "one"
```

and you compare them with the "==" operator, the comparison evaluates to *false* and not to *true*, as you might expect. The reason is that while George and Martha have the same value, they are not the same variable since they point to different memory locations, and therefore are not equal to each other. In functions declared with the *function* keyword, string variables are compared by value, so the actual values of George and Martha are compared. In such cases the result of comparing identical strings with "==" comparison is *true*.

Arrays

An array is a special class of object that refers to its properties with numbers rather than with variable names. Properties of an array object are called elements of the array. The number used to identify an element is called an index and follows an array name in brackets. Array indices must be either numbers or strings.

Array elements can be of any data type. The elements in an array do not all need to be of the same type, and there is no limit to the number of elements an array may have.

The following statements demonstrate assigning values to arrays.

```
var array = new Array;
array[0] = "fish";
array[1] = "fowl";
array["joe"] = new Rectangle(3,4);
array[fool = “creeping things”
array[goo + 1] = “etc.”
```

The variables foo and goo must be either numbers or strings.

Since arrays use a number to identify the data they contain, they provide an easy way to work with sequential data. For example, suppose you wanted to keep track of how many jelly beans you ate each day, so you can graph your jelly bean consumption at the end of the month.

Arrays provide an ideal solution for storing such data.

```
var April = new Array;
April[1] = 233;
April[2] = 344;
April[3] = 155;
April[4] = 32;
```

Now you have all your data stored conveniently in one variable. You can find out how many jelly beans you ate on day x by checking the value of April[x]:

```
for(var x = 1; x < 32; x++)
         Screen.write("On April " + x + " I ate " + April[x] +
            " jellybeans.\n");
```

Arrays usually start at index [0], not index [1]. Note that arrays do not have to be continuous, that is, you can have an array with elements at indices 0 and 2 but none at 1.

Creating arrays

Like other objects, arrays are created using the "new" operator and the Array constructor function. There are three possible ways to use this function to create an array. The simplest is to call the function with no parameters:

```
var a = new Array();
```

This line initializes variable a as an array with no elements. The parentheses are optional when creating a new array, if there are no arguments. If you wish to create an array of a predefined size, pass variable a the size as a parameter of the Array() function. The following line creates an array with a length of the size passed.

```
var b = new Array(31);
```

In this case, an array with length 31 is created.

Finally, you can pass a number of elements to the Array() function which creates an array containing all of the parameters passed. For example:

```
var c = new Array(5, 4, 3, 2, 1, "blast off");
```

creates an array with a length of 6. *c*[0] is set to 5, *c*[1] is set to 4, and so on up to *c*[5], which is set to the string "blast off". Note that the first element of the array is *c*[0], not *c*[1].

Arrays may also be created dynamically. By referring to a variable with an index in brackets, a variable is created as or converted to an array. Arrays created in this manner are unable to use the methods and properties described below, so it is recommended that you use the Array() constructor function to create arrays.

Methods and properties of arrays

When an array is created with the Array() constructor function, a number of methods and properties become available to it.

Properties of arrays

.length

The .length property returns one more than the largest index of the array. Note that this value does not necessarily represent the actual number of elements in an array, since elements do not have to be contiguous.

For example, suppose we had two arrays "ant" and "bee", with the following elements:

```
var ant = new Array;        var bee = new Array;
ant[0] = 3                  bee[0] = 88
ant[1] = 4                  bee[3] = 99
ant[2] = 5
ant[3] = 6
```

The .length property of both ant and bee is equal to 4, even though ant has twice as many actual elements as bee does.

By changing the value of the length property, you can remove array elements. For example, if you change ant.length to 2, ant will only have the first two members, and the

values stored at the other indices will be lost. If we set bee.length to 2, then bee will consist of two members: bee[0], with a value of 88, and bee[1], with an undefined value.

Methods of arrays

.join()

The `.join()` method creates a string of all of array elements. The method has an optional parameter, a string which represents the character or characters that will separate the array elements. By default, the array elements will be separated by a comma. For example:

```
var a = new Array(3, 5, 6, 3);
var string = a.join();
```

will set the value of "string" to "3,5,6,3". You can use another string to separate the array elements by passing it as an optional parameter to the .join() method. For example,

```
var a = new Array(3, 5, 6, 3);
var string = a.join("*/*");
```

creates the string "3*/*5*/*6*/*3".

.sort([compareFunction])

The .sort() method sorts members of an array and puts them in alphabetic order. If no compare function is supplied, then elements are converted to strings to do the conversion, which may cause some confusion. For example, the following code:

```
var a = new Array(32, 5, 6, 3)
a.sort();
var string = a.join();
```

creates a string "3, 32, 5, 6".

This behavior is often not what you want in a sort function. Fortunately, the .sort() method allows you to specify a different way to sort the array elements. The name of the function you want use to compare values is passed as the only parameter to sort().

If a compare function is supplied, the array elements are sorted according to the return value of the compare function. If a and b are two elements being compared, then:

- If compareFunction(a, b) is less than zero, sort b to a lower index than a.
- If compareFunction(a, b) returns zero, leave a and b unchanged to each other.
- If compareFunction(a, b) is greater than zero, sort b to a higher index than a.

By specifying the following function as a sort function, you will get the desired result when comparing numbers:

```
function compareNumbers(a, b)
{
        return a - b
}
```

.reverse()

The reverse() method switches the order of the elements of an array, so that the last element becomes the first.

The following code:

```
ar array = new Array;
array[0] = "ant";
array[1] = "bee";
array[2] = "wasp";
array.reverse();
```

produces the following array:

```
array[0] == "wasp"
array[1] == "bee"
array[2] == "ant"
```

Objects

Variables and functions may be grouped together in one variable and referenced as a group. A compound variable of this sort is called an object in which each individual item of the object is called a property. In general, it is adequate to think of object properties, which are variables or constants, and of object methods, which are functions.

To refer to a property of an object, use both the name of the object and of the property, separated by the operator ".", a period. Any valid variable name may be used as a property name. For example, the code fragment below assigns values to the width and height properties of a rectangle object and calculates the area of a rectangle and displays the result:

```
var Rectangle;

Rectangle.height = 4;
Rectangle.width = 6;

Screen.write(Rectangle.height * Rectangle.width);
```

The main advantage of objects occurs with data that naturally occurs in groups. An object forms a template that can be used to work with data groups in a consistent way. Instead of having a single object called Rectangle, you can have a number of Rectangle objects, each with their own values for width and height.

Predefining objects with constructor functions

A constructor function creates an object template. For example, a constructor function to create Rectangle objects might be defined like the following.

```
function Rectangle(width, height)
{
       this.width = width;
       this.height = height;
}
```

The keyword "this" is used to refer to the parameters passed to the constructor function and can be conceptually thought of as "this object." To create a Rectangle object, call the constructor function with the "new" operator:

```
var joe = new Rectangle(3,4)
var sally = new Rectangle(5,3);
```

This code fragment creates two rectangle objects: one named joe, with a width of 3 and a height of 4, and another named sally, with a width of 5 and a height of 3.

Constructor functions create objects belonging to the same class. Every object created by a constructor function is called an instance of that class. The examples above create a Rectangle class and two instances of it. All of the instances of a class share the same properties, although a particular instance of the class may have additional properties unique to it. For example, if we add the following line:

```
joe.motto = "ad astra per aspera";
```

we add a motto property to the Rectangle joe. But the rectangle sally has no motto property.

Methods - assigning functions to objects

Objects may contain functions as well as variables. A function assigned to an object is called a method of that object.

Like a constructor function, a method refers to its variables with the "this" operator. The following fragment is an example of a method that computes the area of a rectangle.

```
function rectangle_area()
{
       return this.width * this.height;
}
```

Because there are no parameters passed to it, this function is meaningless unless it is called from an object. It needs to have an object to provide values for this.width and this.height.

A method is assigned to an object as the following lines illustrates.

```
joe.area = rectangle_area;
```

The function will now use the values for height and width that were defined when we created the rectangle object joe.

Methods may also be assigned in a constructor function, again using the *this* keyword.

For example, the following code:

```
function rectangle_area()
{
	return this.width * this.height;
}

function Rectangle(width, height)
{
	this.width = width;
	this.height = height;
	this.area = rectangle_area;
}
```

creates an object class Rectangle with the rectangle_area method included as one of its properties. The method is available to any instance of the class:

```
var joe = new Rectangle(3,4);
var sally = new Rectangle(5,3);

var area1 = joe.area();
var area2 = sally.area();
```

This code sets the value of area1 to 12, and the values of area2 to 15.

Object prototypes

An object prototype lets you specify a set of default values for an object. When an object property that has not been assigned a value is accessed, the prototype is consulted. If such a property exists in the prototype, its value is used for the object property.

Object prototypes are useful for two reasons: they ensure that all instances of an object use the same default values, and they conserve the amount of memory needed to run a script. When the two Rectangles, joe and sally, were created in the previous section, they were each assigned an area method. Memory was allocated for this function twice, even though the method is exactly the same in each instance. This redundant memory waste can be avoided by putting the shared function or property in an object's prototype. Then all instances of the object will use the same function instead of each using its own copy.

The following fragment shows how to create a Rectangle object with an area method in a prototype.

```
function rectangle_area()
{
	return this.width * this.height;
}

function Rectangle(width, height)
{
	this.width = width;
	this.height = height;
}
```

```
Rectangle.prototype.area = rectangle_area;
```

The rectangle_area method can now be accessed as a method of any Rectangle object as shown in the following.

```
var area1 = joe.area();
var area2 = sally.area();
```

You can add methods and data to an object prototype at any time. The object class must be defined, but you do not have to create an instance of the object before assigning it prototype values. If you assign a method or data to an object prototype, all instances of that object are updated to include the prototype.

If you try to write to a property that was assigned through a prototype, a new variable will be created for the newly assigned value. This value will be used for the value of this instance of the object's property. All other instances of the object will still refer to the prototype for their values. If, for the sake of this example, we assume that joe is a special Rectangle, whose area is equal to three times its width plus half its height, we can modify joe as follows.

```
function joe_area()
{
	return (this.width * 3) + (this.height/2);
}
joe.area = joe_area;
```

This fragment creates a value, which in this case is a function, for joe.area that supercedes the prototype value. The property sally.area is still the default value defined by the prototype. The instance joe uses the new definition for its area method.

for . . . in

The *for . . . in* statement is a way to loop through all of the properties of an object, even if the names of the properties are unknown. The statement has the following form.

```
for (property in object)
{
	DoSomething(object[property]);
}
```

where object is the name of an object previously defined in a script. When using the for . . . in statement in this way, the statement block will execute once for every property of the object. For each iteration of the loop, the variable property contains the name of one of the properties of object and may be accessed with "object[property]". Note that properties that have been marked with the DONT_ENUM attribute are not accessible to a *for . . . in* statement.

with

The with statement is used to save time when working with objects. It lets you assign a default object to a statement block, so you need not put the object name in front of its properties and methods. The object is automatically supplied by the interpreter.

The following fragment illustrates using the Clib object.

```
with (Clib)
{
       printf("I am a camera");
       srand();
       xxx = rand() % 5;
       putchar(xxx);
}
```

The Clib methods: Clib.printf(), Clib.srand(), Clib.rand(), and Clib.putchar(), in the sample above are called as if they had been written with Clib prefixed. All code in the block following a with statement seems to be treated as if the methods associated with the object named by the with statement were global functions. Global functions are still treated normally, that is, you do not need to prefix "global." to them unless you are distinguishing between two like-named functions common to both objects.

If you were to jump, from within a with statement, to another part of a script, the with statement would no longer apply. In other words, the with statement only applies to the code within its own block, regardless of how the interpreter accesses or leaves the block.

You may not use a goto statement or label to jump into or out of the middle of a with statement block.

Dynamic objects

ScriptEase allows for direct access to the interior workings of how object properties are called. If you wish, you may specify how an object accesses its data by replacing one of the following routines which are internal to ScriptEase. The following methods are available for modifying how an object calls its members. In all cases, the parameter, *property*, is the name of the property being called.

._get(property, ExpectCall)

Whenever the value of a property is accessed, the ._get() method is called. By defining a new ._get() method for an object, you modify the way it accesses property values.

The 4.20 _get function now receives a second parameter. This parameter is called "ExpectCall" and is *true* if the parameter is being retrieved to make a function call, and *false* for other situations.
For example, in this case:

```
obj.foo;
```

The second parameter will be false. But in this case

```
obj.foo();
```

the second parameter will be true.

The example following modifies the Rectangle object created earlier with a new ._get() method. Whenever you access the value of one of the object's properties, it will inform you if the Rectangle is a square. After the object is initialized, the main() function creates an instance of the object with the *width* and *height* properties both set to 3. When the value of the Rectangle.area() method is retrieved, used in a Clib.printf() statement, the dynamic ._get() function is called, which displays, "The rectangle is a square," since *width* and *height* are equal.

```
function rectangle_area()
{
   return this.width * this.height;
}

function rectangle_get(property)
{
   if (this.width == this.height)
       Clib.printf("The rectangle is a square.");
  return this [property];
}

function Rectangle(width, height)
{
   this.width = width;
   this.height = height;
   this._get = rectangle_get;
}

Rectangle.prototype.area = rectangle_area;

main()
{
   var rect = new Rectangle(3, 3);
   Clib.printf("The area of the rectangle is %d.",
               rect.area());
   Clib.getch();
}
```

._put(property, value)

This method controls the way that new data is assigned to a property.

._canPut(property)

This method returns a boolean value indicating whether the property can be written to or not, that is, whether it is read-only or not. For example, you could modify this property to notify users when they try to change read-only values.

._hasProperty(property)

This method returns a boolean value indicating whether or not a property exists.

._delete(property)

This method is called whenever a property is deleted with the delete operator. The property will be "_delete" when the object itself is being deleted.

._defaultValue(hint)

This method returns the primitive value of a variable.

The parameter hint should be either a string or a number that indicates the preferred data type to return. If hint is a string, the method will return a string if possible, otherwise a different type. The actual value of hint is ignored.

._construct(. . .)

This method is called whenever a new object is created with the new operator. The object will have been already created and passed as the this variable to the .construct() method.

._call(. . .)

The call function is called whenever an object method is called. Whatever parameters are passed to the original function will be passed to the call() function.

The following example creates an Annoying object that beeps whenever it retrieves the value of a property.

```
function myget(prop)
{
       System.beep();
       return this[property];
}

var Annoying = new Object;

Annoying.get = myget;
```

Note that the System.beep() method is used only for this example and must be explicitly created for actual use.

._operator(op,operand)

Operator Overloading

ScriptEase allows you to overload the standard arithmetic operators when used with your objects. Consider this example:

```
var a = obj + 10;
```

If 'obj' is one of your own objects, you may have some special meaning you'd like the addition operator to have when applied to it. Operator overloading allows this to be done.

Whenever an object is the first operand to an arithmetic operation, it has the opportunity to redefine what that operation means.
All of the arithmetic operators can be overloaded, such as +, -, /, >>, and so forth.
In addition, the unary operators (i.e. those that have only one operand, the object) can also be overloaded. These are the operators ~, !, ++, --, + , and -.
Finally, the assignment operator (=) can also be overloaded.

Please note that the compound assignment operators (i.e., *=, +=, etc) are treated exactly like you wrote out the statement. In other words,

```
a += b;
```

is treated just like:

```
a = a + b;
```

Overloading the operators will work that way. In this case, if 'a' is an object with overloaded operators, that statement will involve two operators, a '+' and an '='.

To overload operators on a particular object, you simply give the object the method '_operator'. This works like all of the other dynamic object methods. For instance, you can put the '_operator' method in a prototype so that all objects of that class inherit the operator overloading. Here is an example:

```
function overload(op,operand)
{
   Clib.printf("overloading occurring on operator
               '%s'\n",op);
   return DYN_DEFAULT;
}

var myObject = new Object();
myObject._operator = overload;

myObject = 10;
```

The operator overloading function is passed two parameters. The first parameter is the operator itself, in the form of a string. It will be "+" or "-" or "++", etc. The second parameter is the second operand to the operator.
If the object operation being overloaded is 'obj + 4', for instance, then the first parameter is "+" and the second parameter is the number 4. The unary operators (such as '-obj') do not have a second operand, so the second parameter is undefined. You can use this to

distinguish the operators + and - which can be used either way, i.e. the difference between 'obj + 4' and '+obj'.

Whatever value the operator function returns is taken to be the value of the expression. If the operator function returns DYN_DEFAULT or OPERATOR_DEFAULT_BEHAVIOR, then the normal operation is done.
In many cases, you will not want to override all of the operators that could be applied to your object, so you will return this value if the operator is not one you are interested in. In the example above, we print out a message when the object is used in an operation, but we don't change what the operation does. We always return DYN_DEFAULT and thus do the normal ECMAScript operation.

The global object and its properties

Global variables are members of the global object. To access global properties, you do not need to use an object name. For example, to access the isNaN() method, which tests to see whether a value is equal to the special value NaN you can call either of the following.

```
isNaN(value);
```

or

```
global.isNaN(value);
```

The exception to this rule occurs when you are in a function that has a local variable with the same name as a global variable. In such a case, you must use the global keyword to reference the global variable.

Properties of the global object

._argc

This property refers to the number of parameters passed to the main() function of a script. The name of the script is always the first parameter, so if `._argc == 1`, then the script received no arguments. See the main() function for more information on argc and the main() function.

._argv

This property is an array of strings. Each string is a parameter passed to the script's main() function. The value of argv[0] is always the name of the script being called. The first parameter passed to the script is in argv[1]. See the main() function for more information on argc, argv, and the main() function.

Methods of the global object

.eval(expression)

This method evaluates whatever is represented by the parameter expression. If expression is not a string, it will be returned. For example, calling eval(5) returns the value 5.

If expression is a string, the interpreter tries to interpret the string as if it were JavaScript code. If successful, the method returns the last variable with which was working, for example, the return variable. If the method is not successful, it returns the special value, undefined.

.parseInt(string [, radix])

This method converts an alphanumeric string to an integer number. The first parameter, string, is the string to be converted, and the second parameter, radix, is an optional number indicating which base to use for the number. If the radix parameter is not supplied, the method defaults to base 10 which is decimal. If the first digit of string is a zero, radix defaults to base 8 which is octal. If the first digit is zero followed by an "x", that is, "0x", radix defaults to base 16 which is hexadecimal.

Whitespace characters at the beginning of the string are ignored. The first non-whitespace character must be either a digit or a minus sign (-). All numeric characters following the string will be read, up to the first non-numeric character, and the result will be converted into a number, expressed in the base specified by the radix variable. All characters including and following the first non-numeric character are ignored. If the string is unable to be converted to a number, the special value NaN will be returned.

.parseFloat(string)

This method is similar to parseInt() except that it reads decimal numbers with fractional parts. In other words, the first period, ".", in the parameter string is considered to be a decimal point, and any following digits are the fractional part of the number. The method .parseFloat() does not take a second parameter.

.escape(string)

The .escape() method receives a string and escapes the special characters so that the string may be used with a URL. All uppercase and lowercase letters, numbers, and the special symbols, @ * + - . /, remain in the string. All other characters are replaced by their respective Unicode sequence.

.unescape(string)

This method is the reverse of the .escape() method and removes escape sequences from a string and replaces them with the relevant characters.

.isNaN(number)

This method returns *true* if the parameter, *number*, evaluates to NaN, Not a Number. Otherwise it returns *false*.

.isFinite(number)

This method returns *true* if the parameter, number, is or can be converted to a number. If the parameter evaluates as NaN, Number.POSITIVE_INFINITY, or Number.NEGATIVE_INFINITY, the method returns *false*.

Exception Handling via Scripts

First for script code, exceptions are trapped with try:

```
try
{
    do something;
}
catch( e )
{
    Clib.printf("Something bad happened:
%s\n",e.toString();
}
```

A catch clause 'eats' the error, so the rest of the script continues. If you 'throw' something, that something is passed up the chain as an error. You can throw the error object you caught in a catch statement to make the error be 'unhandled'.

For instance:

```
try
{
    do something;
}
catch( e )
{
    Clib.printf("Something bad happened:
%s\n",e.toString();
    throw e;
}
```

In this case, if there is an error, it will be printed out, but then the program will still stop with that error.

You can raise arbitrary errors as you like in a program, i.e.:

```
throw new TypeError("You are not my type!");
```

A try block can also have a finally clause, e.g.:

```
try
{
    do something;
}
finally
{
    Clib.printf("Always happens.\n");
}
```

The finally clause ALWAYS is executed right before the block is left, even if left by a goto, return, error, or whatever. If the finally block does a control transfer (i.e. it does a goto, throw, or return), that takes precedence, else whatever transfer was pending actually does happen.

So if you do:

```
try
{
    return 10;
}
finally
{
    Clib.printf("BYE!\n");
}
```

This will print BYE! then return 10 from the function. If you do:

```
try
{
    return 10;
}
finally
{
    goto no_way;
}

no_way:  ...
```

In this case, the goto takes precedence over the return, so the return is ignored and execution continue with the '...' code.

Preprocessing

This section describes directives that affect the processing of a ScriptEase script prior to finally compiling, tokenizing, and executing the script.

Preprocessor Directives

The following ScriptEase statements that begin with a # character are collectively called *preprocessor directives*, since they are processed before a script is actually executed and direct the way the script commands are interpreted. Preprocessor directives can only be used with the ScriptEase interpreter. Other JavaScript interpreters will not recognize them.

#define

The #define directive is used to replace a token or almost any identifier with other characters. The #define directive is executed while the script is being read into the

interpreter, before the script itself is executed. The #define directive causes one string to be replaced by another in the script that goes to the interpreter. All substitutions are made before the code is interpreted. A #define directive has the following structure.

```
#define token replacement
```

This line results in all subsequent occurrences of "token" being replaced by "replacement". Consider the following line.

```
#define NumberOfCountriesInSouthAmerica 13
```

The define statement increases program legibility and makes it easier to change code later. If Bolivia and Peru decide someday to unite, you only have to change the #define statement to update your program. Otherwise, you would have to go through your script looking for all occurrences of the number 13, decide when they refer to the number of countries in South America, and change them to the number 12.

Likewise, if you write screen routines for a 25-line monitor, and then later decide to make it a 50-line monitor, you're better off altering the following #define directive from:

```
#define ROW_COUNT 25
```

to

```
#define ROW_COUNT 50
```

and using ROW_COUNT in your code. You only have to make one change in your script instead of many.

#include

The #include directive lets you include other scripts, and all of the functions contained therein, as a part of the code you are writing. Usually #include lines are placed at the beginning of the script and consist only of the #include statement and the name of the file to be included, as in the following.

```
#include <gdi.jsh>
#include "gdi.jsh"
#include 'gdi.jsh'
```

Any one of these lines make all of the functions in the library file gdi.jsh available to the script that has the line. The quote characters, ' or ", may be used in place of the angled brackets < and >.

To include several files in one program simply use multiple #include directives as shown.

```
#include <screen.jsh>
#include <keyboard.jsh>
#include <init.jsh>
#include <comm.jsh>
```

The ScriptEase interpreter will not include a file more than once, so if a file has already been included, a second or subsequent #include directive has no effect. ScriptEase ships with a large number of libraries of pre-written functions that you can use. Library files are plain text files, as are all ScriptEase scripts, and have the extension .jsh as a default.

Since these libraries are external to ScriptEase, they are less static than the standard function libraries, and can be easily expanded or modified as the need arises. The most

recent versions of .jsh libraries are listed on the Nombas downloads page at the following web site:

www.nombas.com

#if, #ifdef, #elif, #else, #endif

These directives are all *preprocessor conditionals* and allow you to specify a different set of script source based on different conditions at run time. Conditional directives are frequently used in scripts designed to run on different operating systems by ensuring that scripts include files that are appropriate for the operating system being used.

#if is used like an *if* statement. *#else* corresponds to an *else* statement. *#elif* corresponds to an *else if* statement. These directives define which block of code will actually be used when a script is interpreted and executed. You must use them with terminating *#endif* directives to mark the ends of code blocks.

```
var fullPathOfFile = Clib.rsprintf("%s\\%s\\%s\\%s",
```

For example, suppose you have a script that builds long path names from directories supplied to it in different variables. If you are working in a DOS-based environment, the backslash character is used to separate directories, so you could indicate the full path of a file in DOS as follows:

```
                    rootdirectory, subdirectory1,
                        subdirectory2, filename);
```

If you ported this script to a UNIX machine, however, you would run into problems since UNIX uses forward slashes to separate directories.

You can get around this problem by defining the separator character differently for each operating system:

```
#if defined(_UNIX_)
       #define PathChar '/'
#elif defined(_MAC_)
       #define PathChar ':'
#else
       #define PathChar '\\'
#endif
```

By putting the separator character in a variable, you can make the script work on any operating system:

```
var fullPathOfFile = Clib.rsprintf("%s%c%s%c%s%c%s",
rootdirectory,
                  PathChar, subdirectory1,
                  PathChar, subdirectory2,
                  PathChar, filename);
```

The *#ifdef* directive is another limited form of #if that is equivalent to "*#if!defined(var)*".

#link

The #link command incorporates pre-compiled libraries, such as dynamic link library (.dll) files, into the ScriptEase interpreter. The #link directive is similar to the #include statement with no parameters. For example, the directive

```
#link "sesock"
```

lets the interpreter use the functions for TCI/IP socket communication. #link takes no parameters other than the name of the library being linked.

Although you could write these functions in JavaScript, the functions in the #link libraries are processor intensive and run much more quickly from a compiled source.

Nombas supplies many #link libraries, such as:

GD	for generating .gif files and other graphics functions
ODBC	for working with ODBC databases
OLEAUTOC	for doing OLE automation
REGEXPSN	to perform complex searches
SESOCK	for working with sockets

Contact Nombas for more information on the #link developer's kit, which lets users to create customized #link libraries. The most recent versions of #link libraries are listed on the Nombas downloads page on our web site:

www.nombas.com

Integrating Language Objects & Libraries

Description and location of the libraries

Nombas provides a suite of libraries for use within your application. These libraries provide a set of useful functions which can be integrated with the ISDK. Each library is a group of functions that perform similar tasks or are otherwise related in some way.

There are three main libraries: the C compatibility library, the ScriptEase library, and the Operating System specific libraries. The source for each library can be found in the SRCLIB directory, which contains a group of directories, each representing a specific library. Below is a layout of the SRCLIB directory and a description of each of the libraries found within SRCLIB\ .

LANG	The language extension library. This contains common global functions such as define() and getArrayLength() which serve as general extensions to the JavaScript language.
ECMA	ECMA compatibility library. This is perhaps the most important library, as it provides the set of objects required by the ECMAScript specification. This includes the String object, Math object, Date object, and all other objects and methods required by the ECMA specification.
CLIB	A C-compatibility library. This library contains the complete set of C library functions in a ScriptEase form for use within scripts. They are all grouped under the 'Clib' object, so there are functions such as Clib.strcmp() and Clib.fopen(). Currently this is the only way to perform file I/O from scripts.
SELIB	A group of Nombas supplied functions which provide useful interfaces for some common tasks. They are grouped under the 'SElib' object. Some functions include SElib.directory() and SElib.dynamicLink().
TEST	A small suite of functions for testing scripts and the language. Grouped under the 'Test' object.
GD	A GIF manipulation library based on the freely available 'gd' package. Most often used as a link library.

MD5	An MD5 library, used for managing checksums. Most often used as link library.
REGEXP	A regular expression library based on the corresponding GNU library. Most often used as a link library.
UUCODE	A library of UU Encoding and Decoding functions. Most often used as a link library.
SESOCK	A socket library for Internet communications. Most often used as a link library.
DSP	Nombas' Distributed Scripted Protocol for communicating between scripts. Most often used as a link library.
IDSP	The internet-enabled version of the DSP library. Most often used as a link library.
WIN	Useful Windows 3.1/95/NT functions. Grouped under SElib object.
OS2	Useful OS2 functions. Grouped under SElib object.
UNIX	Useful Unix functions. Grouped under the SElib object.
DOS	Useful DOS & Win16 functions. Grouped under the SElib object.
MAC	Useful Macintosh functions. Grouped under the SElib object.
NLM	Useful Netware functions.

Five Steps to using the libraries within your application

1: **Add the necessary files** to your project

2: **Include the necessary files** in your jseopt.h file

3: **Define values** to include the appropriate functions

4: **Load the libraries** within your application

5: **Add any application services** to the context that the libraries may use

Step 1: Add the necessary files to your project

Each library requires all of the files within the libraries directory; for example, to include the ECMA libraries you must add the files from the SRCLIB\ECMA directory. You must also add all of the files in the SRCLIB\COMMON directory. All code within these libraries is conditionally compiled into the application based on the choices you will make in step 3. At this point no code has been added to your application.

Step 2: Include the necessary files in your jseopt.h file

Each library has a corresponding header file that is required by the library source files. This header has the name se##libname## (e.g., seecma.h), except for the 'SElib' library, whose header file is simply 'selib.h'. Note that if you simply include the 'seall.h' header file, then this entire process is taken care of for you based on the define values you choose in step 3.

Step 3: Define values to include the appropriate functions

To include the functions in the source files, you must define certain values to enable these functions. These defines have the format:

```
JSE_##LIBNAME##_##FUNCNAME##
```

Some examples are JSE_SELIB_DYNAMICLINK, JSE_ECMA_PARSEFLOAT, and JSE_CLIB_PUTS. These functions must be defined as either 1 or 0. A value of 1 enables the function, and a value of 0 disables the function. Disabling a function comes in useful because there is a form of these defines which by default includes all the functions in the appropriate library. These can then be turned off by defining specific functions to be 0. To include all of the functions within a library, simply define JSE_##LIBNAME##_ALL.

For example, if you wanted to include all of the objects and functions from the ECMA library except for the Date and Buffer objects, and none of the functions from the Clib library except for the Clib.printf() function, you could define these in your jseopt.h file:

```
#define JSE_ECMA_ALL          /* All of the ECMA library */
#define JSE_ECMA_DATE      0  /* But not Date object */
#define JSE_ECMA_BUFFER    0  /* And not Buffer object */
#define JSE_CLIB_PRINTF    1  /* Also include Clib.printf() */
```

After you have made all of the appropriate defines, and included the files in the incjse directory, you must then include the file 'selibdef.h', found in the srcmisc directory. This is an automatically generated header file which will convert your set of defines, including _ALL specifications and disabled functions, into a simple form for use by the libraries. Once this file has been included, a function is either defined or not defined. One can simply check for existence, like #if defined(JSE_SELIB_DYNAMICLINK). Also note that if you include the "seall.h" header file, then selibdef.h is automatically included.

Step 4: Load the libraries within your application

This is the step that adds the functions to your executing context. Once you have initialized a context with jseInitializeExternalLink (or if in a jseAppLinkFunc with a new context), then you should initialize all of the appropriate files. All of the Load functions have the same form:

```
jsebool LoadLibrary_##LibName##(jseContext jseContext);
```

For example, there is the LoadLibrary Lang() function and the LoadLibrary_SElib() function. These functions return a boolean value indicating whether the library was successfully loaded into the context. Once these functions are called, any functions that you have defined through the step above will now be available in the context passed to the functions. There is an alternative function, LoadLibrary_All() which has the same form as the other functions, and will load every library that you have defined any functions in using the rules described above in step 3.

Step 5: Add any application services to the context that the libraries may use

The language libraries have been designed to be completely separate from the application, so that any library may be compiled as a late-binding link library which still has access to the services of the application it is linked to. To accomplish this you must add 'Application Services' which perform application-specific tasks that are needed by the application. There are reasonable defaults within the application, but if you want any special behavior you should add new services. Even if you are compiling the libraries as an internal library, you still must provide these services. Nombas provides a default implementation of these services, which are found in the 'srcapp' directory.

A good example of these services is the Console I/O service. Any library function that needs to read from or write to the system console will do so via the Console I/O service that your application provides. For example, the Clib file I/O calls that read or write from stdin, stdout, or stderr will not go to the underlying library file functions, because that may not be appropriate for your application. Instead, you provide an application service which has a set of functions which get called by the library.

List of Application Services

Here is a list of the different application services available and which libraries use them:

ConsoleIO	Provides Console I/O. A structure with several predefined fields for reading and writing to the console. Used by the Clib and Screen libraries. The default implementation simply uses the C library calls, which should work on all non-Windowing environments, but if you have special needs then look in srcapp\secon.c for a sample Console I/O package. If you are including this file, then use the AddStandardServiceConsoleIO() function call when setting up the interpreting context.
Interpret	Provides a filter function when doing SElib.interpret() or SElib.interpretInNewThread(). The main purpose of this is to do menial tasks such as changing the title of windows if you are in a Windowing environment. By default, nothing special is done before the interpret. A default implementation is found in srcapp\seinterp.c. To include this version call the function AddStandardServiceInterpret() when initializing the context.
Suspend	Provides a suspend function for use with SElib.suspend(). Each OS has a different method of performing suspends, and this Application service gives a method of doing this. The default action is to remain in a while() loop for the specified time, which means that the system is essentially frozen during a suspend. There is a default version in srcapp\sesuspen.c which performs the appropriate actions for each OS. If you are including this file, then call the function AddStandardServiceSuspend() when initializing the interpreting context.
Spawn	Provides a spawn function for use with SElib.spawn(). This is a complicated function for Operating Systems which don't support the standard spawn() function. Therefore it is provided as an Application service. The default action is to do nothing and have the function fail. SElib.spawn() will not function without this application service. An implementation is provided in srcapp\sespawn.c as a starting point for your application. If you include this file, then call the function AddStandardService_Spawn() to add the Application service.

Environment	Provides a mechanism for interacting with the system environment. Because the C library environment calls are non-portable, and different systems represent the environment in different ways, this service gets called by the Clib.putenv() and Clib.getenv() functions. An implementation is provided in srcapp\seenv.c. By default, no action is taken, effectively rendering the environment useless. If you are including the provided file, then call the function AddStandardService_Environment() to add the Environment service.
Errno	Simple value for sharing errno between application and library. Because libraries can optionally be compiled as a DLL, this is the only way to maintain a common errno value between them. By default, errno is simply returned, but this may not be the same errno value as used by the application. Generally, the use of errno is limited to begin with, so it should be used only if necessary. No file needs to be included, you must simply call AddStandardService_Errno() when initializing the context.
MacCWD	Provides a context-specific current directory. For use on Macintosh systems with the Clib.chdir calls, and when searching in the current directory. Because there is only one global working directory, when there are multiple threads then they can change each other's working directory. This service allows you to have one working directory associated with a context. By default the cwd() command is called, but if you are doing multithreading, or you want your application to have its own working directory, then use this service. An implementation is found in srcapp\semaccwd.c. If you are including this file, call AddStandardService_MacCWD() to add this service.
RedirectionInfo	Provides a mechanism for keeping track of re-opened files. Only needed if using Clib.freopen(). By default, the C library freopen() is called, but when these files are closed then it is ambiguous what happens when you write to the old file. Also, this service allows the library to tell if a file has been reopened, so that Console I/O functions will not be called on stdout or stderr if those files have been reopened. The appropriate implementation is found in srcapp\filelist.c. If you are including this file, then call AddStandardService_FileRedirection() to add the structure to the context.

Example

The following examples demonstrates using these Nombas libraries in an application. Let's say that you wish to include the following functions:

- All of the Test library
- All of the Clib library except for freopen(), getenv(), and putenv().
- All of the Lang library
- SElib.interpret() and SElib.dynamicLink()

First we include ALL of the files in the following directories under srclib: COMMON, TEST, CLIB, LANG, and SELIB. We now have the files added to our project, so we must include the necessary header files. For this example we will use the 'seall.h' header file which will include all of the necessary header files for us. Now we must set up the define mechanisms to include all of the appropriate files. For our example, the jseopt.h header file will look like this:

```
/* jseopt.h - ISDK options for our application */

#define JSE_TEST_ALL             /* All of the Test library */
#define JSE_CLIB_ALL             /* All of the Clib library */
#define JSE_CLIB_FREOPEN      0  /* But not Clib.freopen() */
#define JSE_CLIB_GETENV       0  /* And not Clib.getenv() */
#define JSE_CLIB_PUTEN        0  /* Not Clib.putenv() */
#define JSE_LANG_ALL             /* All of the language library*/
#define JSE_SELIB_INTERPRET   1  /* SElib.interpret() */
#define JSE_SELIB_DYNAMICLINK 1  /* SElib.dynamicLink() */

#include "seall.h"   /* Include all the necessary files */
```

Now the only responsibility is adding these libraries and any necessary application services. Our fictitious application has a special console I/O application service. Though we won't show the source here, we can take the example console I/O service found in srcapp\secon.c and modify it to suit our needs. We will assume this has been done, and that the new function is AddMyService_ConsoleIO(). We will also include the Nombas provided suspend function in srcapp\sesupen.c and the Errno service.

Therefore we have a function that looks like the following:

```
jsebool
InitializeMyContext(jseContext jseContext)
{
   /* First we add our application services */
   AddMyService_ConsoleIO(jseContext);
   /* These other two are Nombas-provided */
   AddStandardService_Suspend(jseContext);
   /* found in srcapp\sesuspen.c */
   AddStandardService_Errno(jseContext);

   /* Now we load the libraries */
   if( !LoadLibrary_Test(jseContext) )
      jseLibErrorPrintf(jseContext,
                        "Error initializing Test library\n");
   else if( !LoadLibrary_Lang(jseContext) )
      jseLibErrorPrintf(jseContext,
                        "Error initializing Language library\n");
   else if( !LoadLibrary_Clib(jseContext) )
      jseLibErrorPrintf(jseContext,
                        "Error initializing Clib library\n");
   else if( !LoadLibrary_SElib(jseContext) )
      jseLibErrorPrintf(jseContext,
                        "Error initializing SElib library\n");
}
```

You call this function after the context is created with jseInitializeExternalLink. All of these libraries are now ready to be used. Please note that you could have used a single LoadLibrary – All() call to load the standard library. In addition, an libraries which are unique to your application must still be loaded with jseLoadLibrary().

Compiling libraries as link libraries

Each of these libraries can also be compiled as a link library with a minimal amount of effort. To build a link library, setup your project to compile as a DLL, Shared Object, Code Fragment, or whatever is appropriate for you operating system. Then follow steps 1-3 from above.

The only difference is that you must define JSE_TOOLKIT_LINK instead of JSE_TOOLKIT_APP. This signifies that you are compiling a link library. The only symbols that you need to export from a library are 'jseLoadExtension' and 'jseExtensionVer'. If you are using solely one library, like just Clib (or some subset of Clib), then your work is done. There is a default ExtensionLoadFunc() function in the Clib source which will automatically load the Clib library.

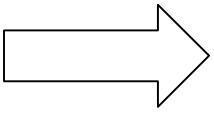
If, however, you want to include more than one library, or part of your own library, then you must write your own load function. To disable the default ones, first define JSE_NO_AUTO_INIT. Then create a function similar to the following in a new file:

```
jsebool FAR_CALL
ExtensionLoadFunc(jseContext jseContext)
{
   jsebool success;

   success = LoadLibrary_Clib(jseContext);
   if( success )
      success = LoadLibrary_SElib(jseContext);

     return success;
}
```

In this example we used both the Clib and SElib libraries, so we needed our own ExtensionLoadFunc. It must be named as 'ExtensionLoadFunc' in order to be called correctly. Once you have compiled successfully, you may then link this library to your application through a '#link' statement in script.



Preprocessor Options: Compile-Time Flags

The ScriptEase:ISDK can be compiled in many ways to suit many different needs. The following is a list of those compile-time, preprocessor defines that determine the options that are built into the ISDK binary libraries and applications.

An advanced user, prepared to modify the default behavior of the ISDK, can change one or more of these options to get just the combination of speed vs. memory vs. standards vs. extensions that they want. We have tested many combinations at Nombas, but not all (the number of possible permutations is tremendous).

Note that in most cases the same value must be defined in the ISDK core, at compile time, as in the application.

Name	**__JSE_DOS16__** or **__JSE_DOS32__** or **__JSE_OS2TEXT__** or **__JSE_OS2PM__** or **__JSE_WIN16__** or **__JSE_WIN32__** or **__JSE_WINCE__** or **__JSE_CON32__** or **__JSE_NWNLM__** or **__JSE_UNIX__** or **__JSE_390__** or **__JSE_MAC__** or **__JSE_PALMOS__** or **__JSE_PSX__**
Purpose	One of these must be defined to describe the target operating system. A few of these are not exclusive, for example when building for Windows CE you should define __JSE_WIN32__ and __JSE_WINCE__.
Default	Default depends on settings in your compiler

Name	**__JSE_DLLLOAD__** or **__JSE_DLLRUN__** or **__JSE_LIB__**
Purpose	One (and only one) of these must be defined for the method for linking with the ISDK. #define __JSE_LIB__ (most common) if the application links directly with the ISDK. #define __JSE_DLLLOAD__ (less common) if the application will dynamically link with the ISDK at the time the application loads. This is common for Windows systems. #define __JSE_LIB__ (rare) in the unusual case where the ISDK is dynamically loaded, but will be specifically loaded at run-time.
Default	#define __JSE_LIB__

Name	**JSETOOLKIT_APP** or **JSETOOLKIT_CORE**
Purpose	One (and only one) of these must be defined to show if building the ISDK CORE or building an application that uses the core. Define JSETOOLKIT_CORE when compiling the files in SRCCORE that make up the core interpreter. Define JSETOOLKIT_APP when compile the application that will like with the core interpreter.
Default	#define JSETOOLKIT_APP

Name	**JSE_FLOATING_POINT**
Purpose	Define whether the native number is a floating-point value. The ECMAScript standards defines an ECMA number as a 64-bit floating point number. This option allows the default to be an integer instead, which means that no floating-point will be supported. Removing floating-point can make the code smaller and faster. #define JSE_FLOATING_POINT 1 is floating-point numbers are to be supported (jsenumber will be a floating-point value). #define JSE_FLOATING_POINT 0 for no floating-point support (jsenumber will be an integer).
Default	#define JSE_FLOATING_POINT 1

Name	**JSE_UNICODE**
Purpose	Define whether Unicode is supported in this implementation (i.e., whether jsechar will be a Unicode or an ASCII character). If Unicode is specified then all elements of strings will be two-byte Unicode values; if not specified then strings are composed of 1-byte ASCII bytes. The ECMAScript standard specifies that all strings will be Unicode, but most ISDK users choose ASCII either to save space and complexity, or because the system they are linking with does not support Unicode. #define JSE_UNICODE 1 for Unicode strings #define JSE_UNICODE 0 for ASCII strings
Default	#define JSE_UNICODE 1 for __JSE_WINCE__ #define JSE_UNICODE 0 for all else

Name	**BYTE_ORDER** or **__BYTE_ORDER** or **BIG_ENDIAN**
Purpose	These defines are used to specify whether the underlying processor is big-endian or little-endian. Various compilers use different means to specify this information, and it is usually handled automatically. If support is wrong for your system then adjust the related defining code in JSETYPES.H
Default	None - usually handled automatically by the compiler

Name	**JSE_POINTER_SIZE** (also **JSE_POINTER_SINT** and **JSE_POINTER_UINT**)
Purpose	JSE_POINTER_SIZE defines the number of bits needed for a pointer (i.e., the number of bits needed to represent a memory address). This has been tested with values of 32 (for most systems) and 16 (for DOS small-memory models). Using a smaller size results in smaller code, but of course you should not use a size different from the memory model of your underlying system. JSE_POINTER_SINT and JSE_POINTER_UINT are used to define how a pointer will be cast to an integer, in those cases where casting is necessary. These values are set automatically for JSE_POINTER_SIZE 8, 16, or 32.
Default	#define JSE_POINTER_SIZE 32

Name	**JSE_POINTER_SINDEX** and **JSE_POINTER_UINDEX**
Purpose	These values define the size of integer needed to index into arrays. Usually these integers will be the same as JSE_POINTER_SINT and JSE_POINTER_UINT (see JSE_POINTER_SIZE) but in some cases you can make them smaller if indexes into arrays will not span the full size needed. For example, if a pointer size is 32-bits but you know that there will never be more than 64K addressed in that memory, then these value can be set to smaller integers to increase performance and to save memory.
Default	For DOS16 or WIN16 if JSE_NO_HUGE is defined (see JSE_NO_HUGE below) #define JSE_POINTER_SINDEX sint #define JSE_POINTER_UINDEX uint For all other cases #define JSE_POINTER_SINDEX slong #define JSE_POINTER_UINDEX ulong

Name	**_NEAR_**, **_FAR_**, **NEAR_CALL**, and **FAR_CALL**
Purpose	In systems that distinguish between near memory and far memory, and near calls and far calls, these values are sometimes used to specify memory use. These values are usually set by default and you don't have to modify them.
Default	For most systems these become comments. See JSETYPES.H for exceptions.

Name	**JSE_NO_HUGE**
Purpose	For DOS16 and WIN16, and other systems that need special calls to allocate inter-segment memory, the ISDK and related libraries will usually call special routines when data falls outside the size of a default allocation segment (see HUGE_MEMORY in UTILHUGE.H). If JSE_NO_HUGE is defined then these special routines will not be used, saving memory use and processing time, but memory sizes larger than a segment will not be permitted, or may cause errors. #define JSE_NO_HUGE if
Default	This value is not defined by default.

Name	**JSE_TOKENSRC** and **JSE_TOKENDST**
Purpose	These define whether the interpreter can create or execute pre-compiled scripts. #define JSE_TOKENSRC 1 if the script can compile plain ECMAScript text into an executable token stream (see jseCreateCodeTokenBuffer()). #define JSE_TOKENSRC 0 if a pre-compiled token buffer cannot be created with this core. #define JSE_TOKENDST 1 if this ISDK build can interpret pre-compiled tokens (via the third parameter to jseInterpret()). #define JSE_TOKENDST 0 if this script cannot interpret pre-compiled token streams Note: The term "token" is a holdover from earlier version of ScriptEase:ISDK in which this really did preserve interpretation at the code at the "token" stage. Beginning with 4.02, a second pass produces True virtual machine opcodes, and it is these opcodes that are saved and executed in these "token" buffers.
Default	#define JSE_TOKENSRC 1 #define JSE_TOKENDST 1

Name	**JSE_SECUREJSE**
Purpose	This defines whether the security guard mechanism is enabled in this version of the ISDK core. A small amount of memory can be saved, and performance gained, if the security manager is not enabled, at the expense of losing this mechanism for preventing rogue scripts from making dangerous calls. #define JSE_SECUREJSE 1 to enable the security manager #define JSE_SECUREJSE 0 to disable the security manager
Default	#if defined(__JSE_DOS16__) # define JSE_SECUREJSE 0 #else # define JSE_SECUREJSE 1 #endif

Name	**JSE_C_EXTENSIONS**
Purpose	Define whether the "C-like" extensions that Nombas has added to the ECMAScript language should be allowed. These extensions allow for the cfunction keyword for function, in which parameters can be passed be reference (instead of by value), and in which strings and buffers are mutable, bytes can be specified individually, and array math is allowed on strings, buffers, and arrays. #define JSE_C_EXTENSIONS 1 to allow C-like extensions for any declared cfunction #define JSE_C_EXTENSIONS 0 to disallow C-like extensions, and stick to the ECMAScript behavior in all functions
Default	#define JSE_C_EXTENSIONS 1

Name	**JSE_LINK**
Purpose	This value defines whether run-time linking of ScriptEase libraries is enabled. These are the libraries added to a script at run time with the #link directive. This is not part of the ECMAScript standard. #define JSE_LINK 1 to enable run-time linking with the #link directive. #define JSE_LINK 0 to disable #link handling
Default	#define JSE_LINK 0 for systems without a default run-time linking method #define JSE_LINK 1 for systems with a default run-time linking method

Name	**JSE_INCLUDE**
Purpose	This directive defines whether run-time inclusion of script files within other script files is enabled. This file-within-a-file inclusion is available with the #include directive. This is not part of the ECMAScript standard. #define JSE_INCLUDE 1 to enable run-time inclusion with the #include directive #define JSE_INCLUDE 0 to disable #include handling
Default	#define JSE_INCLUDE 1

Name	**JSE_DEFINE**
Purpose	This defines whether #define directive will be recognized by the interpreter. #define acts much like the C-language version for text replacement, but does not define macro substitution. This is not part of the ECMAScript standard. #define JSE_DEFINE 1 to enable the #define directive #define JSE_DEFINE 0 to disable #define handling
Default	#define JSE_DEFINE 1

Name	**JSE_CONDITIONAL_COMPILE**
Purpose	This defines whether the interpreter/compiler will recognize C-like preprocessor conditionals, which are #if, #elif, #else, #endif, #ifdef, and #ifndef. The conditionals are not part of the ECMAScript standard. #define JSE_CONDITIONAL_COMPILE 1 to allow conditional preprocessing #define JSE_CONDITIONAL_COMPILE 0 for no conditional
Default	#define JSE_CONDITIONAL_COMPILE 1

Name	**JSE_TOOLKIT_APPSOURCE**
Purpose	This defines whether the interpreter can read ECMAScript source text in a file-like way. If source given to jseInterpret() is only in-memory then this need not be defined. #define JSE_TOOLKIT_APPSOURCE 1 to read source for the file-like routines #define JSE_TOOLKIT_APPSOURCE 0 to disallow source file reading
Default	#define JSE_TOOLKIT_APPSOURCE 1

Name	**JSE_PROTOTYPES**
Purpose	This defines whether ECMAScript "inheritance" is enabled through the prototype mechanism, in which case if an object does not contain the referred-to method or property then its ._prototype chain will be searched. This is standard ECMAScript behavior, and very useful, but can add memory and slow-down performance a small amount. #define JSE_PROTOYPES 1 to allow ._prototype inheritance #define JSE_PROTOYPES 0 to turn off inheritance through ._prototype
Default	#define JSE_PROTOTYPES 1

Name	**JSE_DYNAMIC_OBJS**
Purpose	This allows intrinsic behaviors of objects to be exposed for overridden, so that behavior such as ._get, ._put, ._construct, ._delete, and a few more, can be implemented in object-oriented ways by the underlying C-code or ECMAScript code. This is not technically part of ECMAScript but it is how many of the pre-defined ECMAScript objects are implemented and it's very useful, while adding only a small amount to additional memory usage and performance loss. #define JSE_DYNAMIC_OBJS 1 for dynamic object behavior #define JSE_DYNAMIC_OBJS 0 for no exposition of intrinsic methods
Default	#define JSE_DYNAMIC_OBJS 1

Name	**JSE_API_ASSERTLEVEL**
Purpose	This defines the level of parameter-checking performed by the API-level functions. A higher level of error-checking adds a little more code and memory usage, but prevents common errors by API users. #define JSE_API_ASSERTLEVEL 0 for no parameter checking #define JSE_API_ASSERTLEVEL 1 to check most parameters against *NULL* #define JSE_API_ASSERTLEVEL 2 to check parameters for *NULL*, check against other commons errors, and insert and validate magic "cookies" within valid allocated structures.
Default	#if defined(__JSE_DOS16__) # define JSE_API_ASSERTLEVEL 0 #else # define JSE_API_ASSERTLEVEL 2

Name	**JSE_API_ASSERTLEVEL**
	#endif

Name	**JSE_API_ASSERTNAMES**
Purpose	This parameter defines whether function names will be included in error messages triggered by JSE_API_ASSERTLEVE>0. #define JSE_API_ASSERTNAMES 1 to include names of API functions in error messages #define JSE_API_ASSERTNAMES 0 to prevent API names from being included in error messages
Default	#if defined(__JSE_DOS16__) # define JSE_API_ASSERTNAMES 0 #else # define JSE_API_ASSERTNAMES 1 #endif

Name	**JSE_COMPILER**
Purpose	This defines whether the compiler portion of the interpreter is enabled. Without the interpreter enabled the core interpreter can become smaller by 1/3 to 1/2. #define JSE_COMPILER 1 to allow compilation from plain-text script source #define JSE_COMPILER 0 to disallow compilation, only interpreting pre-compiled code Note: Many of the other preprocessor directive are not sensible if JSE_COMPILER is not enabled. #error messages on compilation will alert to any incompatibilities.
Default	#define JSE_COMPILER 1

Name	**JSE_INLINES**
Purpose	This options is used where performance can improve by using inline functions (e.g. C macros). The drawback is that these inlines can use more memory. #define JSE_INLINES 1 to use inlines (macros) instead of function calls, and get some performance improvements #define JSE_INLINES 0 to use function calls instead of inlines (macros), and use a little less memory
Default	#if defined(__JSE_DOS16__) || defined(__JSE_WIN16__) # define JSE_INLINES 0 #else # define JSE_INLINES 1 #endif

Name	**JSE_MIN_MEMORY**
Purpose	In places where the core interpreter may be coded one way to improve performance, but another way to reduce memory usage, this define can specify which option to favor. #define JSE_MIN_MEMORY 1 to favor small size #define JSE_MIN_MEMORY 0 to favor performance
Default	#if defined(__JSE_DOS16__) # define JSE_MIN_MEMORY 1 #else # define JSE_MIN_MEMORY 0 #endif

Name	**JSE_TYPE_BUFFER**
Purpose	This defines whether jseTypeBuffer will be a valid native data type. ECMAScript does not include any buffer type. #define JSE_TYPE_BUFFER 1 to allow the jseTypeBuffer native type #define JSE_TYPE_BUFFER 0 for no native jseTypeBuffer
Default	#define JSE_TYPE_BUFFER 1

Name	**JSE_CREATEFUNCTIONTEXTVARIABLE**
Purpose	This defines whether a compiled function can be "decompiled" to a string variable. To do so requires saving the tokens before compilation, which can require a lot of memory that is used for no other purpose. #define JSE_CREATEFUNCTIONTEXTVARIABLE 1 to create full-source text from a compiled function #define JSE_CREATEFUNCTIONTEXTVARIABLE 0 to create minimal stub text from compiled functions
Default	#if defined(__JSE_DOS16__) # define JSE_CREATEFUNCTIONTEXTVARIABLE 0 #else # define JSE_CREATEFUNCTIONTEXTVARIABLE 1 #endif

Name	**JSE_GETFILENAMELIST**
Purpose	Defines whether the names of files are preserved for a call to jseGetFileNameList(). A small amount of memory is saved by not preserving this information. #define JSE_GETFILENAMELIST 1 to allow jseGetFileNameList() #define JSE_GETFILENAMELIST 0 to remove jseGetFileNameList()
Default	#if defined(__JSE_DOS16__) # define JSE_GETFILENAMELIST 0 #else # define JSE_GETFILENAMELIST 1 #endif

Name	**JSE_BREAKPOINT_TEST**
Purpose	Define whether the jseBreakpointTest() is enabled for debugging help. This option uses a small amount of memory. #define JSE_BREAKPOINT_TEST 1 to enable jseBreakpointTest() #define JSE_BREAKPOINT_TEST 0 to disable file/line breakpoint testing
Default	#if defined(__JSE_DOS16__) # define JSE_BREAKPOINT_TEST 0 #else #define JSE_BREAKPOINT_TEST 1 #endif

Name	**JSE_MEM_DEBUG**
Purpose	When this option is enabled, the interpreter will check for invalid memory usage. It slows down execution considerably and is recommended only for initial development. If this option is enabled, the interpreter will monitor all memory allocated by calls to jseMalloc(), jseMustAlloc() , jseMustRealloc() and jseRealloc() and freed with calls to jseMustFree(). jseMustAlloc() and jseMustRealloc() will abort if the interpreter is unable to allocate memory, while jseMalloc() will return *NULL*, but otherwise these calls are identical to their C counterparts except that the jseContext is passed before the other parameters. By default, JSE_MEM_DEBUG is enabled if NDEBUG is not defined. #define JSE_MEM_DEBUG for debug-level checks on memory use
Default	#if !defined(NDEBUG) #define JSE_MEM_DEBUG 1 #endif

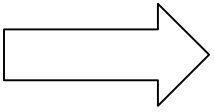
Name	**JSE_FAST_MEMPOOL**
Purpose	This allows many of the smaller structures, which are malloc'ed and freed frequently, to be managed by memory pools. Use of memory pooling for these pointers can speed up runtime execution (approx. 40% in many cases) especially in systems with poorly-performing underlying allocation handlers (such as Win32 DLLs). But use of memory pooling also causes a greater use of memory. #define JSE_FAST_MEMPOOL 1 to use faster memory pooling. #define JSE_FAST_MEMPOOL 0 to turn on memory pooling.
Default	#if defined(__JSE_DOS16__) # define JSE_FAST_MEMPOOL 0 #else # define JSE_FAST_MEMPOOL 1 #endif

Name	**JSE_HASH_STRINGS**
Purpose	Instead of the standard method of string table management, which uses little memory and is permanent, the hash table method uses a greater amount of memory but allows for removal. If this is defined, then entries are removed when they are no longer needed, which is important for people implementing long-running contexts. For example, some users like to keep

Name	**JSE_HASH_STRINGS**
	a single global context. Under the standard method of string table management, every time a new string was added it would remain until the context was destroyed, perpetually increasing memory. The hash table implementation has also been optimized to be faster, and it is now possible to take advantage of the dynamic-object optimization which only works with the hash table.
Default	JSE_HASH_STRINGS 0 if JSE_MIN_MEMORY==1JSE_HASH_STRINGS 1 if JSE_MIN_MEMORY==0

Name	**JSE_ONE_STRING_TABLE**
Purpose	By default the string table is stored in the global call structure. But if this flag is specified a single global static string table is used for all interprets, regardless of contexts or threads.
Default	JSE_ONE_STRING_TABLE 0

Name	**NDEBUG**
Purpose	Most C compilers will use the NDEBUG definition to determine whether extra debugging code will be compiled. This is most commonly used for assert() statements (which are used liberally throughout the ISDK core source), but the ISDK also performs many extra self-debugging functions when NDEBUG is not defined. This extra code provides checks on code integrity, and is important while porting, but has a tremendous hit on footprint and performance.
Default	Many compilers set this on or off based on compiler options



Memory Management

Memory usage can be critical to many applications. SE420 expands on the control of memory that was available in SE410. The biggest change is that we have moved from a reference counting scheme to garbage collection. This was done with no change to the API, so all existing programs will continue to operate unchanged. This guide will explain all of the memory controls available to a ScriptEase ISDK application as well as some tips on optimizing memory performance.

One of the main controls is the JSE_MIN_MEMORY define. By default, only DOS and Windows 16-bit are built with this value on (defined to 1). All other ISDKs are built with it off (defined to 0.) You can override it for your application, like all of the defines mentioned in this section, by predefining it in your compiler. JSE_MIN_MEMORY determines the default goal of the ISDK. If on, the goal is to minimize memory usage, if off it is to minimize execution time. This define determines what default value will be selected for all of the other defines described in this chapter. Of course, you can always override any particular define yourself.

Please note that the rest of this section is necessarily pretty complex, and delves into some of the internals of the ScriptEase engine. You may find it confusing. It is only necessary if memory usage is of critical importance. For most applications on modern systems (with lots of memory), the default settings will work fine. Even for systems low on memory, it is often enough to define JSE_MIN_MEMORY to be 1. I would also suggest looking at the standard ECMA library and turning off some of the lesser-used functions, as they take a lot of code space, especially the regular expression code. The individual settings will only be of interest to the customer who wants exacting control over how memory is used. Only the sections titled 'STRING DATA' and particularly 'OBJECT DESTRUCTORS' will be of interest to most customers, so you might want to skip ahead to them.

The Internal Stack

ScriptEase uses an internal stack for passing parameters to functions as well as calculating the value of expressions used in the program. There are two ways to build the stack. The faster, more memory consuming way is to simply allocate a large block of memory for it. This is the default setting if JSE_MIN_MEMORY is defined to 0. Slower, but more memory efficient, is to grow the stack in pieces as more room is needed. JSE_MIN_MEMORY set to 1 does this. One disadvantage of the faster method is that the stack can run out of memory if the limit you set when compiling is reached. This only matters in highly recursive scripts. The growing stack will continue to function until system memory is exhausted.

You can determine which kind of stack is used by setting JSE_FIXEDSTACK to 1 or 0. If you do turn on the fixed stack, you can also set its size. The define JSE_FIXEDSTACK_DEPTH is the amount of entries that are allocated for it. This value defaults to 10000 entries. Each entry consumes approximately 15 bytes. This is one of the major memory requirements for ScriptEase, so if you are low on memory, we recommend turning the fixed stack off. Each function called needs one entry on the stack per parameter passed to it as well as some entries for any expression it is in the middle of parsing. A typical function will use between 2-10 entries, so expect to be able to recursive to a depth of approximately 1500 with the default stack settings. That should be fine for almost all scripts.

Object Descriptors And Members

In a typical script, the majority of memory will be devoted to storing objects. SE420 uses memory more efficiently than in SE410, namely that important structures are allocated in chunks instead of singularly. Since the system adds an overhead to each allocation (typically 12 bytes), these allocations could use a sizeable fraction of the total memory allocated for overhead, which is now minimized.

Two structures are very important internally, the object descriptor (VarObj) and its members (each described by a Var structure.) Each is usually allocated in chunks of 256 entries. With JSE_MIN_MEMORY on, it is reduced to 128 entries. This define, JSE_VAR_CHUNK_SIZE, can be changed, but it probably will hurt to move it past either extreme. Allocating bigger chunks will result in lots of unused entries. Making each chunk too small means that the overhead as described above will be large in comparison to the chunk's size. I recommend 64 entries as probably a bare minimum size. Note that the growable internal stack described above is a linked-list of these chunks, so this define also determines how much the stack grows with each extension.

Garbage Collection And The Free Lists

Although allocated in chunks, these structures are then moved to a free list individually. When the engine starts, it fills up the free lists by allocating chunks and adding each entry of the chunk onto the appropriate list until it is filled. When any list is empty and an item from that list is required, garbage collection is triggered. The vast majority of collections happen when the Var or VarObj free lists are emptied. The garbage collector does a mark-and-sweep collection. All items currently in use by the engine or locked via the API are marked. Then all items are examined, and those that are not marked are free to be added appropriate free list. Finally, the free lists are 'filled-up' to their maximum number of entries by allocating more chunks. This last part is done because otherwise once most of the items were used, each collection would only free up a few items, meaning collection will be necessary soon again, and this constant collection would drastically slow execution.

The size of these free lists determine how much memory is consumed and how often collection is done. Making the lists big will mean collection happens less frequently, but that more memory is consumed. Making them small will mean much more garbage collections but less wasted memory. However, shrinking them too much isn't recommended, or performance will suffer greatly and very little extra memory will become available. On the other extreme, making them too big will mean each garbage collection pass will need to sweep more memory looking for free items, so if the list sizes are very big, the few memory collections will take lots of time, negating much of the benefits of reducing the number of collections.

The size of these pools in number of elements (not number of chunks) is determined by the defines VAR_POOL_SIZE and VAROBJ_POOL_SIZE. The default values are 4000 and 256 respectively. If JSE_MIN_MEMORY is on, the default values are 256 and 128 respectively. Please note that these pools do not limit the amount of memory used. They are the ideal amount of extra, unused entries kept on hand to be used as needed. Any objects that are in use also consume memory that is not counted in these pools. The ScriptEase engine will keep allocating memory for scripts that try to use a lot of memory (big algorithmically-generated objects come to mind) until memory allocation fails, at which point a fatal error occurs.

Creating and destroying object members will create garbage eventually forcing a collection. However, simple computations on numbers does not create any garbage. Thus, calling a lot of functions (which create and destroy objects internally for each call) will require collection, and be much slower than programs that only do a lot of calculations which is very fast.

String Data

Strings are not pooled, each string entry is allocated as needed and freed when no longer used. This is done because strings are all of variable size. Pools only make sense when each item in the pool is identical. Because there is no pool for string allocation, each string allocated is always allocated from the system. If this was all that was done, garbage collection due to string allocation would never happen until free memory was exhausted, so strings would stick around using up memory unless garbage collection was triggered for some other reason. This, obviously, is not acceptable.

To remedy this, the defined value JSE_STRINGS_COLLECT is the number of bytes of string allocation that will always trigger a garbage collection. Each time the engine has allocated this many bytes of string data, it performs a garbage collection to free up whatever strings are unused. It then resets the counter, ready to garbage collect again when the limit is again reached. Note that this is not how much memory is in-use of

string data, it is simply a counter. There is a good reason it is not how much strings are in use. If it was, once your program had that much strings in use permanently (such as members of objects), every string allocation would then trigger a garbage collection (which would accomplish nothing, since all the strings are still in use) and performance would slow to a crawl.

A good heuristic is to set the value to 1/3 to 1/2 the total amount of memory you would like to 'reserve' for strings. Again, if a script uses a lot of strings that are not freed, the ScriptEase engine will keep allocating space as long it is available. This value just determines how often the engine goes ahead and cleans up whatever strings it has used but are now free. Making this value very high if you have lots of memory will NOT negatively affect performance, unless it is so high that ScriptEase begins to use virtual memory.

When deciding on the value, note that ScriptEase attempts to track how much overhead each string allocation is using, so the value you give should exactly correspond to real memory usage, even if your script allocates a lot of small strings. The default value for JSE_STRINGS_COLLECT is 1000000 (1 million), 100000 if JSE_MIN_MEMORY is on. If your system is on very limited memory, you'll want to shrink this value even more.

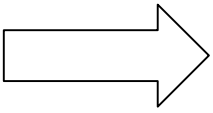
It should be mentioned that if your program is the only one on the system, changing this value is probably pointless. If you run out of memory, garbage collection will be performed, so if there is enough memory to continue, ScriptEase will do so. This define only affects how much memory ScriptEase has allocated but is not using. If you have another program running that can run out of memory because ScriptEase is using it, then this value will be important. If your application is doing its own memory allocation, you can call 'jseGarbageCollect()' if you run out of memory to force the ScriptEase engine to release as much memory as possible, and try again.

Object Destructors

During garbage collection when an object is noted as being freed, and it has a distracter, that object is put on a list. Once garbage collection finishes, those destructors are called. After a destructor is called, the destructor (i.e., the '_delete' property) is removed. This is done because an object can 'resurrect' itself (for instance by assigning itself to a global variable) and thus no longer need to be freed. If the destructor is left around, it can again become free and then resurrect itself, easily leading to an infinite loop. You should not expect to be able to destroy an object more than once.

Since destructors are called when an object is freed, and that only happens in garbage collection, do not expect destructors to be called immediately when the object is no longer in use. Also, do not expect them to be called in any particular order. An API

function is provided to force a garbage collection to occur. This will force all destructors to be called for any objects that are currently free. This API call, jseGarbageCollect(), also allows the collector to be turned off. During this time, no garbage collection is performed, instead whenever memory is used up, new memory is always allocated. This setting will slow performance noticeable, and is only recommended for an application that wants to ensure no garbage collection happens during a particular piece of code. See the API documentation for full details on this call.



MBCS Support In SE 4.20

Nombas ScriptEase™ version 4.20 introduces support for Multibyte Character Sets (MBCS). We have traditionally supported ASCII and Unicode character sets only. Most of the changes involved with supporting MBCS are internal to the ScriptEase engine, and will not affect those using the ISDK. This document will explore the issues that are relevant to the ScriptEase:ISDK user.

In order to use the ScriptEase MBCS support, you will need to define the macro 'JSE_MBCS' to be 1 in your 'jseopt.h' file. In addition, if you are porting to a new operating system or C library you may need to edit the file 'seuni.h' found in the 'srcmisc' directory. SEUNI.H has the defines necessary to call multibyte versions of common functions.

We also make a few minor assumptions about MBCS which you should verify are true for your system. We expect all whitespace characters and the string terminator '\0' to be stored using a single byte. Second, we expect that the second byte of a 2-byte MBCS character will never be the '\0' byte. This allows an MBCS C-style string to be treated as a block of memory terminated by a '\0'. This allows significant performance gains for times when the string must be treated as a whole and we are not interested in its individual characters.

Writing MBCS Compatible Code

If you want to write wrapper functions and ScriptEase API applications that will work in an MBCS environment, there are a number of things you should keep in mind. First, string pointers should not be defined as 'char *' and characters should not be 'char's. Instead, use the 'jsecharptr', ‘jsecharptrdatum’, and 'jsechar' typedefs. Doing so is also necessary for Unicode compatibility and is highly encouraged even if you don't immediately plan to use any character set other than ASCII. A jsechar is the size of the largest individual byte, while a jsecharptrdatum is the size of an individual element as seen by your underlying C library (typical sizes for jsechar/jsecharptrdatum in ASCII:1/1, Unicode:2/2, MBCS:2/1).

You can allocate arrays of jsechars just like you would allocate arrays of chars. This is typically done for a temp storage space when you know a string can never grow past a certain size. These arrays will always be big enough to hold the given number of characters, even if space may be unused depending on what characters it eventually holds.

When accessing individual elements of a string, you should almost never access a string directly using the '[]' or '*' operators, nor can you directly manipulate a string pointer using '++' or '+' or '-'. Instead, you need to use the macros defined in 'seuni.h'. If you use these macros on an ASCII or Unicode build of our engine, they end up doing exactly the above operators, but for MBCS builds they get turned into real function calls.

ScriptEase API Notes

For all API calls in ScriptEase that deal with string lengths and offsets, the API expects logical lengths. This means you pass the number of characters, not the number of bytes. There are only a few routines in which this matters.

The reasoning is that most script-related routines make most sense with logical characters. When you want to use jseSetArrayLength, for instance, you are changing the JavaScript length, i.e. changing the number of characters in the string.

There are a few routines in which both ways make sense. For instance, sometimes when you call jseGetString() you want the length to be in bytes to know how much buffer space was used. In other cases, you want logical characters because you are then going to iterate through the string. We have chosen logical characters for a number of reasons. First compatibility; existing code, for our libraries and written by customers, expects the returns in logical characters, and we would need to rewrite it for MBCS. Second, simplicity: logical characters are the same for all builds, so code correctly written (see above) will work on any character set build of ScriptEase.

Speed And Size

Executables built with MBCS support will be significantly larger than those without. This is because MBCS requires calling functions to do common string manipulation, such as retrieving a character or incrementing a string pointer to point to the next character. These functions are implemented in a regular build by the '[0]' or '*' C operators for retrieving characters and the '++' operator for incrementing a pointer. Obviously, most C compilers will optimize these operations highly, resulting in far less code space than that necessary to call a function.

Performance suffers for the same reason that executable size increases. However, there are a number of places in the core in which performance is many times worse. This is due to the nature of our translation; we have done in se420 a more-or-less direct port of our existing code to support MBCS. While this ensures correctness, it is far from as efficient as possible. We are committed to improving performance in future versions of ScriptEase by finding especially slow performing places in the code and optimizing them. Customers who have a script that shows very poor performance should send that script to Nombas so that we can look at why it is so slow and improve it.

Performance suffers in MBCS builds for the same reasons that executable size increases. We are committed to improving performance in future versions of ScriptEase by finding any especially slow-performing places in the code and optimizing them. Customers who have a script that shows poor MBCS performance should send that script to Nombas so that we can look at how to optimize that section of the core engine for MBCS.

Integrating the ScriptEase Debugger

Using the ScriptEase:ISDK, you can debug your applications using Nombas's debugger. The debugger itself is a Windows application, so you'll need a windows machine to do your debugging on. However, your application can be running on any machine that can communicate with your debugging machine. Nombas provides support for debugging via TCP-IP, but you can extend the debugger to use other communication protocols.

For end-user information on using the debugger, please see the chapter, “Using the ScriptEase Debugger.”

Using a Nombas protocol model

Nombas has provided two models of debugging to cover many situations. First, on windows systems, you can communicate via shared memory. In this case, the debugger and the application must both be running on the same Windows machine. Either the application can start the debugger, or the debugger can start the application (depending on how you set it up.)

If you are debugging using the TCP-IP model, you need to run the ScriptEase IDE Network Extender (called the proxy) on the debugging machine before running your application. The application will communicate with the proxy in place of the debugger. The proxy will make sure the debugger starts up and receives the information it needs to debug your application.

Defining your own protocol model

To define a new protocol model for communication between the debugger and your application requires you to provide a number of routines linked with your application. These are documented at the top of the file 'srcdbg\debugme.h'. You can examine the file 'srcdbg\debugme.c' to see how these routines are implemented in the Nombas-provided models. If you are defining your own model, you will need to also add that model to the proxy.

Code changes to your application

You must do six things to make sure your application is debuggable. They are described in order:

Set Options

```
#define JSE_DEBUGGABLE 1
```

If you are using TCP-IP, set these flags:

```
#define JSE_DEBUG_TCPIP
#define JSE_DEBUG_MASTER
#define JSE_DEBUG_RUN
#define JSE_DEBUG_FILES
#define JSE_DEBUG_REMOTE
#define JSE_DEBUG_PASSWORD
```

(Note: `JSE_DEBUG_PASSWORD` only activates the password code. You must still actually setup a password if you are going to use it.)

Otherwise, if using shared memory set these flags:

```
#define JSE_DEBUG_MEMORY
#define JSE_DEBUG_RUN
```

This setup for shared memory assumes you want the debugger to start the application. If you'd like it to be the other way around (i.e. the application starts the debugger), add:

```
#define JSE_DEBUG_MASTER
```

Add files to your project

Next add the file 'srcdbg\debugme.c' to your application. Make sure the 'srcdbg' directory is in your include path if it isn't already.

Update your ToolkitAppData structure and jseopt.h

If you don't currently allocate one, you must do so. You can get a definition for one by include 'seclib\seseclib.h'. Alternately, if you already are using your own such structure, add the following to it:

```
#if defined(JSE_DEBUGGABLE)
        struct debugMe * debugme;
#endif
```

You need some includes added at the end of your jseopt.h file:

```
#if defined(JSE_DEBUGGABLE)
          #if defined(__JSE_OS2TEXT__) ||
              defined(__JSE_OS2PM__)
              #include <sys\socket.h>
              #include <netinet\in.h>
              #include <netdb.h>
              #include <utils.h>
              #include <nerrno.h>
              #include <sys\ioctl.h>
          #endif
          #include "dbgshare.h"
          #include "proxy.h"
          #include "debugme.h"
#endif
```

Initialize debugging

After you have initialized your external link and added any libraries, but before you start interpretting you must initialize the connection to the debugger. This is done with the following code:

```
#if defined(JSE_DEBUGGABLE)
          debugmeInit(jsecontext,<command line>,<instance>);
#endif
```

The 'command line' is only needed for shared memory debugging if the debugger is going to be starting up your application. It should be the entire command line, which is easily constructed by concatenating the entries of the argv[] array separated by spaces. The routine will extract the debugging command information from the command line. When it returns, you must reparse the command line into individual arguments (which is easily accomplished using strtok().) For the TCP-IP model, the command line parameter is ignored, so you can safely pass NULL.

The 'instance' parameter is the Windows HINSTANCE value for your program. It is only needed if debugging on a Windows platform using a windowed application (as opposed to a console application.) On other platforms, debugmeInit() does not take a third parameter.

Finally, for the TCP-IP version, you must specify what machine will be the debugging

machine. You do this by setting the environment variable 'REMOTE_ADDR' to the machine host name of the debugging machine. You can set this either before launching your program or within your program before calling debugmeInit(). The machine in question needs to have the proxy running as described above.

Call the debugger hook

Finally, you must call the debugger in your MayIContinue function. Here is an example. If you already do some code in your function, do this in addition.

```
            jsebool JSE_CFUNC FAR_CALL
  ContinueFunction(jseContext jsecontext)
  {
            struct ToolkitAppData * SeData =
                ToolkitAppDataFromContext(jsecontext);

            #if defined(JSE_DEBUGGABLE)
                if ( NULL != SeData->debugme )
                {
                    debugmeDebug(SeData->debugme,jsecontext);
                    if ( jseQuitFlagged(jsecontext) )
                        return False;
                }
            #endif

            jsecontext = jsecontext;      /* to prevent warning
                about unused
                                          /* variable */
            return True;
  }
```

Terminate debugging

This code shows you how to terminate debugging. It assumes 'AppData' is a pointer to your application data structure.

```
#   if defined(JSE_DEBUGGABLE)
            {
                struct debugMe *debugme = AppData->debugme;

                if ( NULL != debugme )
                {
                    debugmeHasTerminated(debugme);

                    while ( debugme )
                    {
                         debugmeDebug(debugme,jsecontext);
                         debugme = AppData->debugme;
                       }
                }
                debugmeTerm(jsecontext);
            }
#   endif
```

You must terminate debugging before you destroy the context. You usually terminate debugging right before you exit. This means all scripts you interpret will be debugged in a single session. However, you can terminate then restart debugging if you want each jseInterpret() to be in its own debug session.

Notes

Once you have made these changes, your application can be debugged. You can make all of these changes and still not debug your application if you skip Initialization of debugging. So, if you want, you can only initialize the connection to the debugger if your user selects a special 'debug application' menu item or such.

You can currently only debug scripts that have a filename (i.e. if you tell jseInterpret() to interpret the contents of a file.)

Samples

In seisdk\samples\debug, you can find a modified version of 'simple0' that is debuggable. Run the application from an MS-DOS prompt after first setting 'REMOTE_ADDR' as described above.

Example: Modifying your JSEOPT.H file for debugging

Any application that uses the debugger must have the following lines in its JSEOPT.H file:

```
#define JSE_DEBUGGABLE 1
#define JSE_DEBUG_RUN
```

Set these flags if you will be using the debugger remotely:

```
#define JSE_DEBUG_TCPIP
#define JSE_DEBUG_MASTER
#define JSE_DEBUG_FILES
#define JSE_DEBUG_REMOTE
#define JSE_DEBUG_PASSWORD
```

Set this flag if you will be using the debugger locally:

```
#define JSE_DEBUG_MEMORY
```

With the options described above, the debugger will launch the application in order to debug it. For the reverse, in which the application launches the debugger, define the following:

```
#define JSE_DEBUG_MASTER
```

Security

As a scripting language, ScriptEase provides you with the power to completely control your system. But there are times when this power can be dangerous. Many applications, such as those using ScriptEase's distributed scripting capabilities, may need to run scripts that you do not want to have access to all of ScriptEase's power. You don't want these scripts to delete files on your machine, read important data and transmit it to a remote machine, execute arbitrary system programs, or any other such activities. ScriptEase security allows you to limit scripts so they cannot do these things.

ScriptEase security works by dividing functions on the system into **secure functions,** those which can perform no dangerous actions, and **insecure functions,** those which can perform dangerous activities. When you execute a script, you can attach a security manager to it. This manager will determine which insecure functions can be called.

If the script tries to call an insecure function which the manager does not allow, it will not call the function and instead generate a security error. By using ScriptEase security, you can run scripts you trust and give them full access to dangerous functions, such as Clib.system() and Clib.remove(), while denying access to these same functions to other scripts you don't trust.

Writing a Security Manager

Whenever you wish to interpret a script, via the API using jseInterpret() or in a script using SElib.interpret(), you can attach a security manager to that child script you are running. As long as that child script calls other functions in only within that script, it is allowed to do so. If it tries to call an insecure function, your security gets called. Obviously, insecure wrapper functions are always checked.

In the case that a script is using SElib.interpret() to interpret a child script, that child may be able to try to call functions in the parent. Since the security you added only applies to the child script, the functions in your original script are also considered insecure to the child. The child must get permission to call them exactly like it would need to get permission to call an insecure wrapper function directly.

You can think of your security manager as a big wall with a heavily guarded door. As long as the script stays on its side of the wall, it is fine. The parent script and all wrapper functions are on the other side of the wall. If the child script wants to get access to them, it has to convince the guards to let it through.

Let's look at the pieces that make up these security guards.

jseSecurityInit

This function is the main security function. It is run before the script it is protecting is run, and it sets up the security the child is going to be run under. It specifies which functions the child will be allowed to call. By default, the child will not be allowed to call any insecure functions. In this function, you explicitly specify which insecure functions the child will be allowed to call. You do this by calling the 'setSecurity' function. It is a member function of all ScriptEase functions.

In case that is confusing, a quick example of a jseSecurityInit function should clear it up:

```
function jseSecurityInit(security_var)
{
	Clib.remove.setSecurity(jseSecureAllow);
}
```

This particular security initialization function is written in ScriptEase. However, you can also implement all of these functions using the ScriptEase API and wrapper functions. We will implement the examples as scripts for clarity. The first thing you notice about the function is that it takes a parameter, we have named it 'security_var'. We did not use it in this example. This parameter is the 'security variable' described below.

The body of the function will usually just list which functions are to be allowed. Notice that we call the 'setSecurity' as a member of the particular function we want to allow. This function takes one parameter, the security state of the function. 'jseSecureAllow' specifies that this function is allowed to be called.

There are two other values we could have used instead. 'jseSecureReject' will cause calls to the function to fail. This is the default for all functions, so it is usually redundant to specify it. However, if 'setSecurity()' is called more than once on the same function, the last call takes precedence. You can use this value to undo allowing access to a particular function.

The final value is 'jseSecureGuard' which says that any time this function is called, we must first call the jseSecurityGuard function to determine if the call will be allowed. This function is described below.

> **Note** that the 'setSecurity' member function can only be called in a security initialization function. Trying to call it at other times will generate an error.

jseSecurityTerm

Whenever you have an initialization function, you have a corresponding termination

function. Like 'jseSecurityInit', this function gets a single parameter, the security variable (described below.) This function is rarely needed, and you can simply not specify it most of the time. It is included so that you can clean up the security variable before exiting. You don't need to 'unset' the setSecurity() calls done, as the engine knows that they go away when they are no longer used. The security termination function looks like this:

```
function jseSecurityTerm(security_var)
{
	/* do any necessary cleanup */
}
```

This function is not usually called until the end of the program (not just the end of the script.) Why is this? If you have read the 'advanced concepts' chapter, you know that all of the functions in a jseInterpret() stick around in the global object, even after the jseInterpret() call itself is finished. This is why you can 'load' functions using jseInterpret() and later call them. Whatever security they had when they were created isn't forgotten.

All functions remember the security in effect when they were created, and that applies if they are again called later. So, the security termination function isn't actually called until all of the functions have gone away, which happens at the end of the program when the ScriptEase engine cleans up everything.

jseSecurityGuard

Usually it is enough to specify which functions you want to allow to be called in the jseSecurityInit function and leave it at that. There can be cases in which you want to allow a function to be called with certain parameters but reject it with others. For instance, you may want to limit creating sockets to certain ports or limit opening files to certain filenames. You specify 'jseSecureGuard' for the setSecurity() options for these functions, and before they can be called, your jseSecurityGuard function will first be called to validate this call.

Here is an example:

```
	function jseSecurityGuard(security_var,func,filename)
	{
		if( func==Clib.fopen )
		{
			/* get the full path so the user can't trick
us with
			* something like:
'c:\\temp\\..\\windows\\win.ini'
			*/
			var actualname = SElib.fullpath(filename);

			/* We only want to allow files in this
directory to be opened. */
			return
Clib.strnicmp("c:\\temp\\",actualname,8)==0;
		}
		else
		{
			return false;
		}
	}
```

This function, like the other two, gets the security variable as its first parameter. Again, we will describe that shortly. The second parameter is the actual function being called. In this example, we compare it to 'Clib.fopen' so that we can validate a call to 'Clib.fopen()'. The security guard function must return 'true' to allow the call, 'false' to disallow it. In this case, we return false if it is not Clib.fopen. Presumably, we only label Clib.fopen as 'jseSecureGuard', so only Clib.fopen will be using this guard function.

We include the else clause because it is always a good idea to cover all bases. If it is something we don't expect, we just say no. This is good programming practice in general; if the parameters aren't what you expect, even if you think it is impossible for them not to be, still do something sensible even if that turns out not to be the case.

Notice that this function has a third parameter, 'filename'. All of the parameters that are being passed to the called function are also passed to the security guard function after the two parameters it always gets. The first parameter to the called function is the third to security guard, the second we receive as our fourth, and so on. This allows us to examine the parameters the function will get when deciding if we want to allow the call. In fact, there would be little point in not examining the parameters. If we are always going to reject or accept a particular call regardless of the parameters, we can instead just set that up in the 'jseSecurityInit' function.

Perceptive readers will note that Clib.fopen actually takes two parameters, but we have only named one of them. In Javascript, you can pass extra parameters to script functions,

more than are named in the parameter list. These parameters are still there and can be accessed using the 'arguments' array. In this case, 'filename' is the same as 'arguments[2]', and we could have referred to it that way. The file mode parameter to Clib.fopen() will also be passed to us. We can refer to it as 'arguments[3]', or we can name it in the parameter list if we need to check it as well.

This example checks the name and only allows file access in the 'C:\temp\' directory. We could limit it in any way we choose, this is just one possibility.

securityVariable

We mentioned above that each function gets a security variable passed to it. Each security manager has a single variable associated with it. You can specify this when you specify your security functions (see below for specifying security). Alternately, if you don't, a blank ScriptEase object is created (as if calling 'new Object()') and used. This variable cannot be accessed by the script being run, but it is passed to each security function whenever it is called. This allows you to store data needed to implement your security and keep it safe from the script being run.

Specifying Security

The ScriptEase API call jseInterpret() has among its settings 'jseNewSecurity.' If you turn this on, then the script being run will have security applied to it. If you leave it off, no security applies and all functions can be called. The four security items we just finished discussing correspond to the four fields of the jseExternalLinkParameters structure of the same name. Before you interpret the script, you use jseGetExternalLinkParameters() to get the parameters structure, fill in these fields, then call jseInterpret() with the jseNewSecurity flag turned on. You must fill in the 'jseSecurityInit' function. If you do not, the 'jseNewSecurity' flag will be ignored.

Since the parameters are jseVariables, you set them to any function you like. You can use jseCreateWrapperFunction() to create a wrapper function to do the security tasks. In the example above, we used script examples. **ScriptEase Desktop** implements security this way. The three functions are put in a script. You tell ScriptEase Desktop the name of the script using the command line parameter '/secure=<security script name>'. ScriptEase Desktop interprets that script first, picks out the security functions, and uses them when it interprets the script you are really interested in. The functions in the security script must be given the names we described above.

When you interpret a script from within a script, using SElib.interpret(), you can also specify the security for that child script. See the manual description of SElib.interpret() for details on how you do this.

Wrapper Functions And Security

Wrapper functions are insecure because they are labelled that way. When you write your own wrapper functions and add them using jseAddLibrary(), you get to label them as either secure or insecure. Remember, if there is any possible way the function could be misused, make it insecure. If you are in doubt about whether a particular function should be labelled secure or insecure, choose insecure.

When you are writing a wrapper function, it is possible for it to use jseCallFunction() or jseInterpret() to execute more code. These calls are affected by security. This allows security to propogate. For instance, the ECMAScript function 'eval()' executes a text string as script code exactly like the text string appeared directly in the script. In this case, the wrapper acts just as a passthrough, and the code it executes should follow all of the standard security rules. In fact, the ECMA eval() function itself is secure; whatever text it executes has the same security as what was already executing. ScriptEase uses this model when you use these two API calls. As a result, the following behavior applies:

When calling a function using jseCallFunction(), the call is treated as if the wrapper function's caller was making the call. This means that the calling script function will need to get approval to call the new function. Typically, a wrapper function that just turns around and uses jseCallFunction() is itself secure.

jseInterpret() has different behavior depending on the wrapper function itself. If the wrapper function is insecure, then the script run with jseInterpret() starts with no security. If the wrapper function is secure, then jseInterpret() starts with the same security as the calling function.

So, for instance, ECMA eval() is secure as we already mentioned. Thus, when it runs a new script, that script has the existing security restrictions still on it. If the function was labelled insecure, then it has already passed a security check to be able to call it, and it can continue to do dangerous things, so any scripts it interprets are likewise at this high level of security. jseInterpret() allows security to be added using the 'jseNewSecurity' flag. This is on top of whatever security it already has as specified above.

Sample Script

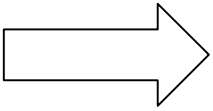
Here is a sample ScriptEase Desktop security script. If you use it, then the desktop scripts will not be allowed to use any insecure functions except a few file-related ones. In addition, Clib.fopen will only be allowed to open files in the 'C:\temp\' directory.

```
	function jseSecurityInit(security_var)
	{
		/* allow basic file manipulations, but nothing fancy,
and
		* make sure to examine all open calls very carefully.
		*/
		Clib.fopen.setSecurity(jseSecureGuard);
		Clib.fclose.setSecurity(jseSecureAllow);
		Clib.fprintf.setSecurity(jseSecureAllow);
		Clib.fread.setSecurity(jseSecureAllow);
		Clib.fwrite.setSecurity(jseSecureAllow);
	}

	function jseSecurityGuard(security_var,func,filename)
	{
		/* we only guard the fopen call, so this should be it
*/
		Clib.assert( security_var==Clib.fopen );

		/* get the full path so the user can't trick us with
something
		* like: 'c:\\temp\\..\\windows\\win.ini'
		*/
		var actualname = SElib.fullpath(filename);

		/* We only want to allow files in this directory to
be opened. */
		return Clib.strnicmp("c:\\temp\\",actualname,8)==0;
	}
```



Language Objects & Libraries

ScriptEase Global Functions

The global functions described in this section are unique to the ScriptEase implementation of JavaScript. In other words, they are not part of the ECMAScript standard, but they are useful. Avoid using these functions in a script if it will be used with a JavaScript interpreter that does not support these unique functions.

Like other global items these functions are actually methods of the global Object and can be called with function or method notation. The two following lines of code are equivalent.

```
var aString = ToString(123)
var aString = global.ToString(123)
```

General

defined(value)

This function tests whether a variable, Object property, or value has been defined. The function returns `true` if a value has been defined, or else returns `false`. The function `defined()` may be used during script execution and during preprocessing. When used in preprocessing with the directive `#if`, the function `defined()` is similar to the directive `#ifdef`, but is more powerful. The following fragment illustrates three uses of `defined()`.

```
var t = 1;
#if defined(_WIN32_)
              Screen.writeln("in Win32");
           if (defined(t))
              Screen.writeln("t is defined");
           if (!defined(t.t))
           Screen.writeln("t.t is not defined");
#endif
```

The first use of `defined()` checks a value available to the preprocessor to determine which platform is running the script. The second use checks a variable "t". The third use checks an object "t.t"

getArrayLength(array[, MinIndex])

This function should be used with dynamically created arrays, that is, with arrays that were **not** created using the Array() constructor and new operator. When working with arrays created using the Array() constructor and new operator, use the .length property of the arrays. The .length property is not available for dynamically created arrays which must use the functions, getArrayLength() and setArrayLength(), when working with array lengths.

The getArrayLength() function returns the length of a dynamic array, which is one more than the highest index of an array, if the first element of the array is at index 0, which is most common. If the parameter MinIndex is passed, then it is used to set to the minimum index, which will be zero or less. You can use this function to get the length of an array that was not created with the Array() constructor function.

This function and its counterpart, setArrayLength(), are intended for use with dynamically created arrays, that is, arrays not created with the Array() constructor function. Use the .length property to get the length of arrays created with the constructor function and not getArrayLength().

getAttributes(variable)

This function gets and returns the variable attributes for the parameter variable. Variable attributes may be set using the function setAttributes(). See setAttributes() for more information and descriptions of the attributes of variables that can be set.

setArrayLength(array[, MinIndex], length])

This function sets the first index and length of an array. Any elements outside the bounds set by MinIndex and length are lost, that is, become undefined. If only two arguments are passed to setArrayLength(), the second argument is length and the minimum index of the newly sized array is 0. If three arguments are passed to setArrayLength(), the second argument, which must be 0 or less, is the minimum index of the newly sized array, and the third argument is the length.

setAttributes(variable, attributes)

This function sets the variable attributes for the parameter variable using the parameter attributes. Variables in ScriptEase may have various attributes set that affect the behavior of variables. This function has no return.

The following list describes the attributes that may be set for variables. Multiple attributes may be set for variables by combining them with the or operator. For example, the flag setting `READ_ONLY | DONT_ENUM` sets both of these attributes for one variable.

DONT_DELETE	This variable may not be deleted. If the delete operator is used with a variable, nothing is done.
DONT_ENUM	This variable is not enumerated when using a for/in loop.
IMPLICIT_PARENTS	This attribute applies only to local functions and allows a scope chain to be altered based on the __parent__ property of the "this" variable. If this flag is set, if the __parent__ property is present, and if a variable is not found in the local variable context, activation object, of a function, then the parents of the "this" variable are searched backwards before searching the global object. The example below illustrates the effect of this flag.
IMPLICIT_THIS	This attribute applies only to local functions. If this flag is set, then the "this" variable is inserted into a scope chain before the activation object. For example, if variable TestVar is not found in a local variable context, activation object, the interpreter searches the current "this" variable of a function.
READ_ONLY	This variable is read-only. Any attempt to write to or change this variable fails.

The following fragment illustrates the use of setAttributes() and the behavior affected by the IMPLICIT_PARENTS flag.

```
function foo()
{
   value = 5;
}
setAttributes(foo, IMPLICIT_PARENTS)

var a;
a.value = 4;
var b;
b.__parent__ = a;
b.foo = foo;
b.foo();
```

After this code is run, a.value is set to 5.

undefine(value)

This function undefines a variable, Object property, or value. If a value was previously defined so that its use with the function defined() returns true, then after using undefine() with the value, defined() returns false. Undefining a value is different than setting a value to null.

In the following fragment, the variable n is defined with the number value of 2, and then undefined.

```
var n = 2;
undefine(n);
```

In the following fragment an object `o` is created and a property `o.one` is defined. The property is then undefined but the object `o` remains defined.

```
var o = new Object;
o.one = 1;
undefine(o.one);
```

Conversion or casting

Though ScriptEase does well in automatic data conversion, there are times when the types of variables or data must be specified and controlled. Each of the following casting functions has one parameter, which is a variable or piece of data, to be converted to or cast as the data type specified in the name of the function. For example, the following fragment creates two variables.

```
var aString = ToString(123);
var aNumber = ToNumber("123");
```

The first variable aString is created as a string from the number 123 converted to or cast as a string. The second variable aNumber is created as a number from the string "123" converted to or cast as a number. Since aString had already been created with the value "123", the second line could also have been:

```
var aNumber = ToNumber(aString);
```

The type of the variable or piece of data passed as a parameter affects the returns of some of the functions.

ToBoolean(value)

The following table lists how different data types are converted by this function.

Data type	Return
Boolean	same as value
Buffer	same as for String
`null`	`false`
Number	`false` if value is 0, +0, -0 or NaN, else `true`
Object	`true`
String	`false` if empty string, "", else `true`
undefined	`false`

ToBuffer(value)

This function converts value to a buffer in a manner similar to ToString() except that the resulting array of characters is a sequence of ASCII bytes and not a unicode string.

ToBytes(value)

This function converts value to a buffer and differs from ToBuffer() in that the conversion is actually a raw transfer of data to a buffer. The raw transfer does not convert unicode values to corresponding ASCII values. For example, the unicode string "Hit" may be stored in a buffer as "\0H\0\i\0t", that is, as the hexadecimal sequence: 00 48 00 69 00 74.

ToInt32(value)

This function is the same as ToInteger() except that if the return is an integer, it is in the range of -2^{31} through $2^{31} - 1$.

ToInteger(value)

This function converts value to an integer type. First, call ToNumber(). If result is NaN, return +0. If result is +0, -0, +Infinity or -Infinity, return result. Else return floor(abs(result)) with the appropriate sign. For example, the value -4.8 is converted to -4.

ToNumber(value)

The following table lists how different data types are converted by this function.

Data type	Return
Boolean	+0, if value is `false`, else 1
Buffer	same as for String
`null`	0
Number	same as value
Object	first, call ToPrimitive(), then call ToNumber() and return result
String	number, if successful, else `NaN`
undefined	`NaN`

ToObject(value)

The following table lists how different data types are converted by this function.

Data type	Return
Boolean	new Boolean object with value
`null`	generate runtime error
Number	new Number object with value
Object	same as parameter
String	new String object with value
undefined	generate runtime error

ToPrimitive(value)

This function does conversions only for parameters of type Object. An internal default value of the Object is returned.

ToString(value)

The following table lists how different data types are converted by this function.

Data type	Return
Boolean	"`false`", if value is `false`, else "`true`"
`null`	"`null`"
Number	if value is NaN, return "NaN". If +0 or -0, return "0". If Infinity, return "Infinity". If a number, return a string representing the number. If a number is negative, return "-" concatenated with the string representation of the number.
Object	first, call ToPrimitive(), then call ToString() and return result
String	same as value
undefined	"undefined"

ToUint16(value)

This function is the same as ToInteger() except that if the return is an integer, it is in the range of 0 through 2^{16} - 1.

ToUint32(value)

This function is the same as ToInteger() except that if the return is an integer, it is in the range of 2^{32} - 1.

The Buffer Object

The Buffer object provides a way to manipulate data at a very basic level. It is needed whenever the relative location of data in memory is important. Any type of data may be stored in a buffer object. A new Buffer object may be created from scratch or from a string, buffer, or Buffer object, in which case the contents of the string or buffer will be copied into the newly created Buffer object. To create a Buffer object, follow the syntax below.

```
new Buffer([size] [, unicode] [, bigEndian]);
```

A line of code following this syntax creates a new buffer object. If size is specified, then the new buffer is created with the specified size, filled with `NULL` bytes. If no size is specified, then the buffer is created with a size of 0, though it can be extended dynamically later. The unicode parameter is an optional boolean flag describing the

initial state of the .unicode flag of the object. Similarly, bigEndian describes the initial state of the bigEndian parameter of the buffer. If unspecified, these parameters default to the values described below.

```
new Buffer( string [, unicode] [, bigEndian] );
```

A line of code following this syntax creates a new buffer object from the string provided. If string is a unicode string (unicode is enabled within the application), then the buffer is created as a unicode string. This behavior can be overridden by specifying `true` or `false` with the optional boolean unicode parameter. If this parameter is set to `false`, then the buffer is created as an ASCII string, regardless of whether or not the original string was in unicode or not. Similarly, specifying `true` will ensure that the buffer is created as a unicode string. The size of the buffer is the length of the string (twice the length if it is unicode). This constructor does not add a terminating `NULL` byte at the end of the string. The bigEndian flag behaves the same way as in the first constructor.

```
new Buffer( buffer [, unicode] [, bigEndian]);
```

A line of code following this syntax creates a new buffer object from the buffer provided. The contents of the buffer are copied as-is into the new buffer object. The unicode and bigEndian parameters do not affect this conversion, though they do set the relevant flags for future use.

```
new Buffer( bufferObject );
```

A line of code following this syntax creates a new buffer object from another buffer object. Everything is duplicated exactly from the other bufferObject, including the cursor location, size, and data.

All of the above calls have an equivalent call form (such as "Buffer(15)"), except that this simply returns the buffer part (equivalent to the data member), rather than the entire Buffer object.

Buffer Object Properties

.size

The size of the Buffer object. This property may be assigned to, such as "foo.size = 5". If a user changes the size of the buffer to something larger, then it is filled with *NULL* bytes. If the user sets the size to a value smaller than the current position of the cursor, then the cursor is moved to the end of the new buffer.

.cursor

The current position within a buffer. This value is always between 0 and .size. It can be assigned to as well. If a user attempts to move the cursor beyond the end of a buffer, than the buffer is extended to accommodate the new position, and filled with *NULL* bytes. If a user attempts to set the cursor to less than 0, then it is set to the beginning of the buffer, to position 0.

.unicode

This property is a boolean flag specifying whether to use unicode strings when calling .getString() and .putString(). This value is set when the buffer is created, but may be changed at any time. This property defaults to the unicode status of the underlying ScriptEase engine.

.bigEndian

This property is a boolean flag specifying whether to use bigEndian byte ordering when calling .getValue() and .putValue(). This value is set when a buffer is created, but may be changed at any time. This property defaults to the state of the underlying OS and processor.

.data

This property is a reference to the internal data of a buffer. It is only a temporary value to assist in passing parameters to OS and system-library type calls. In the future, all ScriptEase library functions should be able to recognize Buffer objects and to get this member on their own.

Buffer Object Methods

.putValue(value [, valueSize] [, valueType])

This method puts the specified value into a buffer. The value must be a number. ValueSize or both valueSize and valueType may be passed as additional parameters. ValueSize is a positive number describing the number of bytes to be used and defaults to 1. Acceptable values for valueSize are 1,2,3,4,8, and 10, providing that it does not conflict with the optional valueType flag. (See listing below.)

The parameter valueType must be one of the following: "signed", "unsigned", or "float". It defaults to "signed." The valueType parameter describes the type of data to be read. Combined with valueSize, any type of data can be put. The following list describes the acceptable combinations of valueSize and valueType:

valueSize	valueType
1	signed, unsigned
2	signed, unsigned
3	signed, unsigned
4	signed, unsigned, float
8	float
10	float (Not supported on every system)

Any other combination will cause an error. The value is put into the buffer at the current cursor position, and the cursor value is automatically incremented by the size of the value to reflect this addition. To explicitly put a value at a specific location while preserving the cursor location, do something similar to the following.

```
var oldCursor = foo.cursor;   // Save the old cursor location
foo.cursor = 20;              // Set to new location
```

```
  foo.putValue(goo);            // Put goo at offset 20
  foo.cursor = oldCursor        // Restore cursor location
```

The value is put into the buffer with byte-ordering according to the current setting of the .bigEndian flag. Note that when putting float values as a smaller size, such as 4, some significant figures are lost. A value such as "1.4" will actually be converted to something to the effect of "1.39999974". This is sufficiently insignificant to ignore, but note that the following does not hold true:

```
foo.putValue(1.4,4,"float");
  foo.cursor -= 4;
  if( foo.getValue(4,"float") != 1.4 )
  // This is not necessarily true due to sig. dig. loss.
```

This situation can be prevented by using 8 or 10 as a valueSize instead of 4. A valueSize of 4 may still be used for floating point values, but be aware that some loss of significant figures may occur (though it may not be enough to affect most calculations).

.getValue([valueSize] [, valueType])

This method returns a value from the specified position in a buffer object. This call is similar to the putValue() function, except that it gets a value instead of puts a value.

.putString(string)

This method puts a string into the buffer object at the current cursor position. If the .unicode flag is set within the Buffer object, then the string is put as a unicode string, otherwise it is put as an ASCII string. The cursor is incremented by the length of the string (or twice the length if it is put as a unicode string). Note that terminating *NULL* byte is not added at end of the string. To put a *NULL* terminated string, the following can be done.

```
  foo. putString("Hello");   // Put the string into the buffer
  foo.putValue( 0 );         // Add terminating NULL byte
```

.getString([length])

This method returns a string starting from the current cursor location and continuing for length bytes. If no length is specified, then the method reads until a *NULL* byte is encountered or the end of the buffer is reached. The string is read according to the value of the .unicode flag of the buffer. A terminating *NULL* byte is not added, even if a length parameter is not provided.

.toString()

This method returns a string equivalent of the current state of the buffer. Any conversion to or from unicode is done according to the .unicode flag of the object.

.subBuffer(beginning, end);

This method returns another Buffer object consisting of the data between the positions specified by the parameters: beginning and end. If the parameter beginning is less than 0, then it is treated as 0, the start of the buffer. If the parameter end is beyond the end of the buffer, then the new sub-buffer is extended with *NULL* bytes, but the original buffer is

not altered. The .unicode and .bigEndian flags are duplicated in the new buffer. The size of the new buffer is set to the beginning and end parameters. If the cursor in the old buffer is between beginning and end, then it is converted to a new relative position in the new buffer. If the cursor was before beginning, then it is set to 0 in the new buffer, and if it was past end, then it is set to the end of the new buffer.

Buffer[offset]

This is an array-like version of the .getValue()/.putValue() methods which works only with bytes. A user may either get or set these values, such as "`goo = foo[5];`" or "`foo[5] = goo;`". Every get/put operation uses byte types, that is, SWORD8. If offset is less than 0, then 0 is used. If offset is beyond the end of a buffer, the size of the buffer is extended with *NULL* bytes to accommodate it.

The Date Object

ScriptEase shines in its ability to work with dates and provides two different systems for working with them. One is the standard `Date` object of JavaScript and the other is part of the `Clib` object which implements powerful routines from C. Two methods, `Date.fromSystem()` and `.toSystem()`, convert dates in the format of one system to the format of the other. The standard JavaScript `Date` object is described in this section.

To create a `Date` object which is set to the current date and time, use the `new` operator, as you would with any object.

```
var currentDate = new Date();
```

There are several ways to create a `Date` object that is set to a date and time. The following lines all demonstrate ways to get and set dates and times.

```
var aDate = new Date(milliseconds);
var bDate = new Date(datestring);
var cDate = new Date(year, month, day);
var dDate = new Date(year, month, day, hours, minutes, seconds);
```

The first syntax returns a date and time represented by the number of milliseconds since midnight, January 1, 1970. This representation by milliseconds is a standard way of representing dates and times that makes it easy to calculate the amount of time between one date and another. Generally, you do not create dates in this way. Instead, you convert them to milliseconds format before doing calculations.

The second syntax accepts a string representing a date and optional time. The format of such contains one or more of the following fields, in any order:

```
month day, year hours:minutes:seconds
```

For example, the following string:

```
"Friday 13, 1995 13:13:15"
```

specifies the date, Friday 13, 1995, and the time, one thirteen and 15 seconds PM, which, expressed in 24 hour time, is 13:13 hours and 15 seconds. The time specification is optional and if included, the seconds specification is optional.

The third and fourth syntaxes are self-explanatory. All parameters passed to them are integers.

year	If a year is in the twentieth century, the 1900s, you need only supply the final two digits. Otherwise four digits must be supplied.
month	A month is specified as a number from 0 to 11. January is 0, and December is 11.
day	A day of the month is specified as a number from 1 to 31. The first day of a month is 1 and the last is 28, 29, 30, or 31.
hours	An hour is specified as a number from 0 to 23. Midnight is 0, and 11 PM is 23.
minutes	A minute is specified as a number from 0 to 59. The first minute of an hour is 0, and the last is 59.
seconds	A second is specified as a number from 0 to 59. The first second of a minute is 0, and the last is 59.

For example, the following line of code:

```
var aDate = new Date(1492, 9, 12)
```

creates a Date object containing the date, October 12, 1492.

The following is a brief description of the methods of the `Date` object. Instance methods are shown with a period, ".", at their beginnings. A specific instance of a variable should be put in front of the period to call a method.
For example, the `Date` object `aDate` was created above, and, to call the `.getDate()` method, the call would be: `aDate.getDate()`. Static methods have "`Date.`" at their beginnings, since these methods are called with a literal call, such as `Date.parse()`. These methods are part of the `Date` object itself instead of instances of the `Date` object.

Instance Date methods

.getDate()

This method returns the day of the month, as a number from 1 to 31, of a `Date` object. The first day of a month is 1, and the last is 28, 29, 30, or 31.

.getDay()

This method returns the day of the week, as a number from 0 to 6, of a `Date` object. Sunday is 0, and Saturday is 6.

.getFullYear()

This method returns the year, as a number with four digits, of a `Date` object.

.getHours()

This method returns the hour, as a number from 0 to 23, of a `Date` object. Midnight is 0, and 11 PM is 23.

.getMilliseconds()

This method returns the millisecond, as a number from 0 to 999, of a `Date` object. The first millisecond in a second is 0, and the last is 999.

.getMinutes()

This method returns the minute, as a number from 0 to 59, of a `Date` object. The first minute of an hour is 0, and the last is 59.

.getMonth()

This method returns the month, as a number from 0 to 11, of a `Date` object. January is 0, and December is 11.

.getSeconds()

This method returns the second, as number from 0 to 59, of a `Date` object. The first second of a minute is 0, and the last is 59.

.getTime()

This method returns the milliseconds representation of a `Date` object, in the form of an integer representing the number of seconds from midnight on January 1, 1970, GMT, to the date and time specified by a `Date` object.

.getTimezoneOffset()

This method returns the difference, in minutes, between Greenwich Mean Time (GMT) and local time.

.getUTCDate()

This method returns the UTC day of the month, as a number from 1 to 31, of a `Date` object. The first day of a month is 1, and the last is 28, 29, 30, or 31.

.getUTCDay()

This method returns the UTC day of the week, as a number from 0 to 6, of a `Date` object. Sunday is 0, and Saturday is 6.

.getUTCFullYear()

This method returns the UTC year, as a number with four digits, of a `Date` object.

.getUTCHours()

This method returns the UTC hour, as a number from 0 to 23, of a `Date` object. Midnight is 0, and 11 PM is 23.

.getUTCMilliseconds()

This method returns the UTC millisecond, as a number from 0 to 999, of a `Date` object. The first millisecond in a second is 0, and the last is 999.

.getUTCMinutes()

This method returns the UTC minute, as a number from 0 to 59, of a `Date` object. The first minute of an hour is 0, and the last is 59.

.getUTCMonth()

This method returns the UTC month, as a number from 0 to 11, of a `Date` object. January is 0, and December is 11.

.getUTCSeconds()

This method returns the UTC second, as number from 0 to 59, of a `Date` object. The first second of a minute is 0, and the last is 59.

.getYear()

This method returns the year, as a number with two digits, of a `Date` object.

.setDate(DayOfMonth)

This method sets the day, as a number from 1 to 31, of a `Date` object to the parameter `DayOfMonth`. The first day of a month is 1, and the last is 28, 29, 30, or 31.

.setFullYear(year[, month[, date]])

This method sets the year of a `Date` object to the parameter year. The parameter year is expressed with four digits.

If the parameter month is passed, use data format for `.setMonth()`.

If the parameter date is passed, use data format for `.setDate()`.

.setHours(hour[, minute[, second[, millisecond]]])

This method sets the hour, as a number from 0 to 23, of a `Date` object to the parameter hours. Midnight is 0, and 11 PM is 23.

If the parameter `minute` is passed, use data format for `.setMinutes()`.

If the parameter `second` is passed, use data format for `.setSeconds()`.

If the parameter `millisecond` is passed, use data format for `.setMilliseconds()`.

.setMilliseconds(millisecond)

This method sets the millisecond, as a number from 0 to 59, of a `Date` object to the parameter millisecond. The first `millisecond` in a second is 0, and the last is 999.

.setMinutes(minute[, second[, millisecond]])

This method sets the minute, as a number from 0 to 59, of a `Date` object to the parameter minute. The first minute of an hour is 0, and the last is 59.

If the parameter `second` is passed, use data format for `.setSeconds()`.

If the parameter `millisecond` is passed, use data format for `.setMilliseconds()`.

.setMonth(month[, date])

This method sets the month, as a number from 0 to 11, of a `Date` object to the parameter month. January is 0, and December is 11.

If the parameter `date` is passed, use data format for `.setDate()`.

.setSeconds(second[, millisecond])

This method sets the second, as a number from 0 to 59, of a `Date` object to the parameter second. The first second of a minute is 0, and the last is 59.

If the parameter millisecond is passed, use data format for `.setMilliseconds()`.

.setTime(milliseconds)

This method sets a `Date` object to the date and time specified by the parameter milliseconds which is the number of milliseconds from midnight on January 1, 1970, GMT.

.setUTCDate(DayOfMonth)

This method sets the UTC day, as a number from 1 to 31, of a `Date` object to the parameter `DayOfMonth`. The first day of a month is 1, and the last is 28, 29, 30, or 31.

.setUTCFullYear(year[, month[, date]])

This method sets the UTC year of a `Date` object to the parameter year. The parameter year is expressed with four digits.

If the parameter `month` is passed, use data format for `.setUTCMonth()`.

If the parameter `date` is passed, use data format for `.setUTCDate()`.

.setUTCHours(hour[, minute[, second[, millisecond]]])

This method sets the UTC hour, as a number from 0 to 23, of a `Date` object to the parameter hours. Midnight is 0, and 11 PM is 23.

If the parameter `minute` is passed, use data format for `.setUTCMinutes()`.

If the parameter `second` is passed, use data format for `.setUTCSeconds()`.

If the parameter `millisecond` is passed, use data format for `.setUTCMilliseconds()`.

.setUTCMilliseconds(millisecond)

This method sets the UTC millisecond, as a number from 0 to 59, of a `Date` object to the parameter `millisecond`. The first millisecond in a second is 0, and the last is 999.

.setUTCMinutes(minute[, second[, millisecond]])

This method sets the UTC minute, as a number from 0 to 59, of a `Date` object to the parameter `minute`. The first minute of an hour is 0, and the last is 59.

If the parameter `second` is passed, use data format for `.setUTCSeconds()`.

If the parameter `millisecond` is passed, use data format for `.setUTCMilliseconds()`.

.setUTCMonth(month[, date])

This method sets the UTC month, as a number from 0 to 11, of a `Date` object to the parameter `month`. January is 0, and December is 11.

If the parameter `date` is passed, use data format for `.setUTCDate()`.

.setUTCSeconds(second[, millisecond]])

This method sets the UTC second, as a number from 0 to 59, of a `Date` object to the parameter `second`. The first second of a minute is 0, and the last is 59.

If the parameter millisecond is passed, use data format for .setUTCMilliseconds().

.setYear(year)

This method sets the year of a `Date` object to the parameter year. The parameter year may be expressed with two digits for a year in the twentieth century, the 1900s. Four digits are necessary for any other century.

.toGMTString()

This method converts a `Date` object to a string, based on Greenwich Mean Time.

.toLocaleString()

This method returns a string representing the date and time of a `Date` object based on the time zone of the user.

.toSystem()

This method converts a `Date` object to a system time format which is the same as that returned by the `Clib.time()` method. To create a Date object from a variable in system time format, see the `Date.fromSystem()` method.

.toUTCString()

This method returns a string that represents the UTC `date` in a convenient and human-readable form.

Static Date methods

The Date object has three special methods that are called from the object itself, rather than from an instance of it: `Date.fromSystem()`, `Date.parse()`, and `Date.UTC()`.

Date.fromSystem(time)

This method converts the parameter time, which is in the same format as returned by the `Clib.time()`, to a standard JavaScript `Date` object. To create a `Date` object from date information obtained using `Clib`, use code similar to:

```
var SysDate = Clib.time();
var ObjDate = Date.fromSystem(SysDate);
```

To convert a `Date` object to system format that can be used by the methods of the Clib object, use code similar to:

```
var SysDate = ObjDate.toSystem();
```

Date.parse(datestring)

This method converts the string datestring to a `Date` object. The string must be in the following format:

```
Friday, October 31, 1998 15:30:00 -0500
```

This format is used by the `.toGMTString()` method and by email and Internet applications. The day of the week, time zone, time specification or seconds field may be omitted.

```
var theDate = Date.parse(datestring);
```

is equivalent to:

```
var theDate = new Date(datestring);
```

Date.UTC(year, month, day, [, hours [,minutes [,seconds]]])

This method interprets its parameters as a date and returns the number of milliseconds from midnight, January 1, 1970, to the date and time specified. The parameters are interpreted as referring to Greenwich Mean Time (GMT).

The Math Object

The Math object in ScriptEase has a full and powerful set of methods and properties for mathematical operations. A programmer has a rich set of mathematical tools for the task of doing mathematical calculations in a script.

Properties

Math.E

The number value for e, the base of natural logarithms. This value is represented internally as approximately 2.7182818284590452354.

Math.LN10

The number value for the natural logarithm of 10. This value is represented internally as approximately 2.302585092994046.

Math.LN2

The number value for the natural logarithm of 2. This value is represented internally as approximately 0.6931471805599453.

Math.LOG2E

The number value for the base 2 logarithm of e, the base of the natural logarithms. This value is represented internally as approximately 1.4426950408889634. The value of `Math.LOG2E` is approximately the reciprocal of the value of `Math.LN2`.

Math.LOG10E

The number value for the base 10 logarithm of e, the base of the natural logarithms. This value is represented internally as approximately 0.4342944819032518. The value of `Math.LOG10E` is approximately the reciprocal of the value of `Math.LN10`.

Math.PI

The number value for pi, the ratio of the circumference of a circle to its diameter. This value is represented internally as approximately 3.14159265358979323846.

Math.SQRT1_2

The number value for the square root of ½, which is represented internally as approximately 0.7071067811865476. The value of `Math.SQRT1_2` is approximately the reciprocal of the value of `Math.SQRT2`.

Math.SQRT2

The number value for the square root of 2, which is represented internally as approximately 1.4142135623730951.

Methods

Math.abs(x)

Returns the absolute value of x. Returns `NaN` if x cannot be converted to a number.

Math.acos(x)

Returns the arc cosine of x. The return value is expressed in radians and ranges from 0 to pi. Returns `NaN` if x cannot be converted to a number, is greater than 1, or is less than -1.

Math.asin(x)

Returns an implementation-dependent approximation of the arc sine of the argument. The return value is expressed in radians and ranges from -pi/2 to +pi/2. Returns `NaN` if x cannot be converted to a number, is greater than 1, or less than -1.

Math.atan(x)

Returns an implementation-dependent approximation of the arc tangent of the argument. The return value is expressed in radians and ranges from -pi/2 to +pi/2.

Math.atan2(x, y)

Returns an implementation-dependent approximation to the arc tangent of the quotient, y/x, of the arguments y and x, where the signs of the arguments are used to determine the quadrant of the result. It is intentional and traditional for the two-argument arc tangent function that the argument named y be first and the argument named x be second. The return value is expressed in radians and ranges from -pi to +pi.

Math.ceil(x)

Returns the smallest number that is not less than the argument and is equal to a mathematical integer. If the argument is already an integer, the result is the argument itself. Returns `NaN` if x cannot be converted to a number.

Math.cos(x)

Returns an implementation-dependent approximation of the cosine of the argument. The argument is expressed in radians. Returns `NaN` if x cannot be converted to a number.

Math.exp(x)

Returns an implementation-dependent approximation of the exponential function of the argument, that is, returns e raised to the power of the x, where e is the base of the natural logarithms. Returns NaN if x cannot be converted to a number.

Math.floor(x)

Returns the greatest number value that is not greater than the argument and is equal to a mathematical integer. If the argument is already an integer, the return value is the argument itself.

Math.log(x)

Returns an implementation-dependent approximation of the natural logarithm of x.

Math.max(x, y)

Returns the larger of x and y. Returns `NaN` if either argument cannot be converted to a number.

Math.min(x, y)

Returns the smaller of x and y. Returns `NaN` if either argument cannot be converted to a number.

Math.pow(x, y)

Returns the value of x to the power of y.

Math.random()

Returns a number which is positive and pseudo-random and which is greater than or equal to 0 but less than 1. This method takes no arguments.

Math.round(x)

Returns the number value that is closest to the argument and is equal to a mathematical integer. x is rounded up if its fractional part is equal to or greater than 0.5 and is rounded down if less than 0.5.

Math.sin(x)

Returns the sine of x, expressed in radians. Returns `NaN` if x cannot be converted to a number.

Math.sqrt(x)

Returns the square root of x. Returns `NaN` if x is a negative number or cannot be converted to a number.

Math.tan(x)

Returns the tangent of x, expressed in radians. Returns `NaN` if x cannot be converted to a number.

The String Hybrid

The String data type is a hybrid that shares characteristics of primitive data types, Boolean and Number, and of composite data types, `Object` and `Array`. The String is presented in this section under two main headings in which the first describes its characteristics as a primitive data type and the second describes its characteristics as an object.

The String as data type

A string is an ordered series of characters. The most common use for strings is to represent text. To indicate that text is a string, it is enclosed in quotation marks. For example, the first statement below puts the string "hello" into the variable `word`. The second sets the variable `word` to have the same value as a previous variable hello:

```
var word = "hello";
word = hello;
```

Escape sequences for characters

Some characters, such as a quotation mark, have special meaning to the interpreter and must be indicated with special character combinations when used in strings. This allows the interpreter to distinguish between a quotation mark that is part of a string and a quotation mark that indicates the end of the string.

The table below lists the characters indicated by escape sequences:

`\a`	Audible bell	
`\b`	Backspace	
`\f`	Formfeed	
`\n`	Newline	
`\r`	Carriage return	
`\t`	Tab	
`\v`	Vertical tab	
`\'`	Single quote	
`\"`	Double quote	
`\\`	Backslash character	
`\0###`	Octal number	(example: '\033' is the escape character)
`\x##`	Hex number	(example: '\x1B' is the escape character)
`\0`	*NULL* character	(example: '\0' is the *NULL* character)
`\u####`	Unicode number	(example: '\u001B' is the escape character)

Note that these escape sequences cannot be used within strings enclosed by back quotes, which are explained below.

Single quote strings

You can declare a string with single quotes instead of double quotes. There is no difference between the two in JavaScript, except that double quote strings are used less commonly by many scripters. In functions declared with the `cfunction` keyword, the difference is more important. For more information, see the section on `cfunction`.

Back quote strings

ScriptEase provides the back quote "`", also known as the back-tick or grave accent, as an alternative quote character to indicate that escape sequences are not to be translated. Any special characters represented with a backslash followed by a letter, such as "\n", cannot be used in back tick strings.

For example, the following lines show different ways to describe a single file name:

```
"c:\\autoexec.bat" // traditional C method
'c:\\autoexec.bat' // traditional C method
'c:\autoexec.bat'  // alternative ScriptEase method
```

Back quote strings are not supported in most versions of JavaScript. So if you are planning to port your script to some other JavaScript interpreter, you should not use them.

Long Strings: Using + to concatenate or join strings

You can use the + operator to concatenate strings. The following line:

```
var proverb = "A rolling stone " + "gathers no moss."
```

creates the variable proverb and assigns it the string "A rolling stone gathers no moss." If you try to concatenate a string with a number, the number is converted to a string.

```
var newstring = 4 + "get it";
```

This bit of code creates newstring as a string variable and assigns it the string "4get it".

The use of the + operator is the standard way of creating long strings in JavaScript. In ScriptEase, the + operator is optional. For example, the following:

```
var badJoke =  "I was standing in front of an Italian "
"restaurant waiting to get in when this guy "
"came up and asked me, \"Why did the "
"Italians lose the war?\" I told him I had "
"no idea. \"Because they ordered ziti"
"instead of shells,\" he replied."
```

creates a long string containing the entire bad joke.

The String as object

Strings have both properties and methods which are listed in this section. These properties and methods are discussed as if strings were pure objects. Strings have instance properties and methods and are shown with a period, ".", at their beginnings. A specific instance of a variable should be put in front of a period to use a property or call a method. The exception to this usage is a static method which actually uses the identifier `String`, instead of a variable created as an instance of `String`. The following code fragment shows how to access the .length property, as an example for calling a String property or method:.

```
var TestStr = "123";
var TestLen = TestStr.length;
```

String properties

.length

The length of a string can be obtained by using the length property. For example:

```
var string = "No, thank you.";
Screen.write(string.length);
```

displays the number 14, the number of characters in the string.

String instance methods

.charAt()

This method returns a character at a certain place in a string. To get the first character in a string, use index 0, as follows:

```
var string = "a string";
string.charAt(0);
```

To get the last character in a string, use:

```
string.charAt(string.length - 1);
```

.charCodeAt(index)

This method returns a number representing the unicode value of the character at position index of a string. Returns NaN if there is no character at the position.

.indexOf(substring [, offset])

This method returns the index of the first appearance of a substring in a string. For example:

```
var string = "what a string";
string.indexOf("a")
```

returns the position, which is 2 in this example, of the first "a" appearing in the string. The method .indexOf() may take an optional second parameter which is an integer indicating the index into a string where the method starts searching the string.

For example:

```
var magicWord = "abracadabra";
var secondA = magicWord.indexOf("a", 1);
```

returns 3, the index of the first "a" to be found in the string when starting from the second letter of the string. Since the index of the first character is 0, the index of second character is 1.

.lastIndexOf(substring [, offset])

This method is similar to .indexOf(), except that it finds the last occurrence of a character in a string instead of the first.

.split([substring])

This method splits a string into an Array of strings based on the delimiters in the parameter substring. The parameter substring is optional and if supplied, determines where the string is split. If no delimiters are specified, the method returns an Array with one element which is the original string.

For example, to create an Array of all of the words in a sentence, use code similar to the following fragment:

```
var sentence = "I am not a crook";
var wordArray = sentence.split(' ');
```

.substring()

This method retrieves a section of a string. For example, to get the first ten characters in string, use something like the following code fragment:

```
var string = "a string with many words in it";
```

```
var substring = string.substring(0, 10);
```

.toLowerCase()
.toUpperCase()

These two methods change the case of a string. `.toLowerCase()` returns a copy of a string with all of the letters changed to lower case. `.toUpperCase()` returns a copy of a string with all of the letters changed to upper case.

String static methods

String.fromCharCode(char1, char2...)

This method returns a string created from the character codes that are passed to it as parameters. The identifier String is used with this static method, instead of a variable name as with instance methods. The arguments passed to this method are assumed to be unicode characters. The following line:

```
var string = String.fromCharCode(0x0041,0x0042)
```

set the variable string to be "AB".

The SElib Object

The methods in the `SElib` Object extend the functionality of JavaScript. Whereas the `Clib` Object extends the power of JavaScript by providing functions from the standard C library, the `SElib` extends power by allowing programmers to work with such things as directories, files, memory, windows, messages, system operations, and script execution.

Memory

Routines that directly manipulate memory, as these routines do, should be used with caution. A programmer should clearly understand memory and the operations of these methods before using them.

.peek(address[, dataType])

This method reads and returns data from a specified address in memory. The address is a pointer to memory. This method is similar to the `Blob.get()` method with the parameter address replacing the parameters BlobVar and offset. If `dataType` is not specified, then `UWORD8` is assumed.

If dataType is supplied, it must be one of the following values:

UWORD8	SWORD8	UWORD16	SWORD16
UWORD24	SWORD24	UWORD32	SWORD32
FLOAT32	FLOAT64	FLOAT80	

(FLOAT80 is not available for some environments)

See `Clib.fread()` for more information on these values.

.pointer(var)

This method returns the address in memory where the data in parameter `var` is stored.

var must be a string or a buffer. BLObs are okay since they are buffers.

For architectures that distinguish between near and far memory addresses, the value returned by .pointer() is a far address.

ScriptEase data is guaranteed to remain fixed at its memory location only if that memory is not modified by a script. So, a pointer is valid only until a script modifies var or until var goes out of scope in a script. Putting data in the memory occupied by var after such a change is dangerous. Whenever data is put into the memory occupied by var, be careful not to put more data than will fit in the memory that var actually occupies.

This method prepares a BLOb that may be passed to a call in the operating system.

```
// Assume there is an OS call that will perform a command
// on a number of files. This call expects to receive a
// C-defined packed structure like this:
// struct
// {
//   unsigned charFileCount;
//   char *Command;      // Command to perform on files
//   char *FileName[1];  // array of pointers to file
// }                     // names

// Now prepare such a structure Blob.put(Data,2,UWORD);
// how many names follow Blob.put(Data,pointer("DEL"),UWORD32);
// set command

Blob.put(Data, 2, UWORD32);
Blob.put(Data, pointer("DEL"), UWORD32);
Blob.put(Data, pointer("C:\\UTL\\DOG"), UWORD32);
Blob.put(Data, pointer("C:\\UTL\\CAT"), UWORD32);
```

.poke(address, data [, dataDescriptor])

This method writes the value contained in data to the memory location specified by address. The parameter address must be a valid memory pointer. This method is similar to the Blob.put() method, with the parameter address replacing the parameters BlobVar and offset.

If dataDescriptor is not specified then UWORD8 is assumed, and data must be a single byte. If data is a buffer, then dataDescriptor must be the length of the buffer. If data is an object, then dataDescriptor must be a description of that object.

If dataDescriptor is supplied, it must be one of the following values:

UWORD8	SWORD8	UWORD16	SWORD16
UWORD24	SWORD24	UWORD32	SWORD32
FLOAT32	FLOAT64	FLOAT80	

(FLOAT80 is not available for some environments)

See Clib.fread() for more information on these values.

Directories and files

.directory([, fileSpec [, subdirs [, incAttr [, reqAttr]]]])

This method returns an Object whose properties are strings containing the names of files that match the path specification supplied by `fileSpec`. If no files match `fileSpec`, the method returns `null`.

By default, the method `.directory()` returns the names of subdirectories and filenames of normal files, read-only files, and files with the archive bit set. The parameter `incAttr` specifies which attributes are considered when including files in the file list that is returned.

The parameter `fileSpec` is any valid file specification that matches the criteria of an operating system. A `fileSpec` may include drive and path information and may use wildcards anywhere that is allowed by an operating system. If this parameter is not supplied, then the default is the current directory that is current for a script.

The parameter `subdirs` is a Boolean that should be set to `true` to search subdirectories or to `false` to limit a search to the directory, only, that is specified by `fileSpec`. The default is `false`.

The parameter `incAttr` allows files to be retrieved based on what attribute flags are set. Supplying one or more of the following flags causes `.directory()` to return only files that have corresponding bits set:

`FATTR_RDONLY`	Read-only file
`FATTR_HIDDEN`	Hidden file
`FATTR_SYSTEM`	System file
`FATTR_SUBDIR`	Directory
`FATTR_ARCHIVE`	Archive file

Flags that do not apply to an operating system are ignored. If 0 is passed, only files with no attribute bits set are returned. If this parameter is not specified, then the default is:

```
FATTR_RDONLY|FATTR_SUBDIR|FATTR_ARCHIVE|FATTR_NORMAL
```

This default flag setting for incAttr provides an example of how to use the or operator to specify more than one flag.

The parameter `reqAttr` is specified in the same way as the flags for `incAttr`. Only files with all of the specified attributes set are returned.

The file list that is returned by `.directory()` is an Object with the following properties:

`.name`	Full file name, including the SearchSpec path.
`.attrib`	File flags, as defined above in IncAttr.
`.size`	Size of file, in bytes.
`.access`	Date and time of last file access in Clib.time format.

`.write` Date and time of last write to file Clib.time format.

`.create` Date and time of file creation Clib.time format.

The following routine lists all files, except subdirectory entries, in the current directory of a script.

```
function ListDirectory(FileSpec)
{
          var FileList = SElib.directory(FileSpec,
              False,~FATTR_SUBDIR)
          if (null == FileList)
               Clib.printf("No files found for search spec
          \"%s\".\n",
                              FileSpec);
          else
                 {
                     var FileCount = 1 +
               getArrayLength(FileList);
                     for (var i = 0; i < FileCount; i++)
                     Clib.printf("%s\tsize = %d\tCreate
               date/time = %s\n",
                     FileList[i].name, FileList[i].size,
                     Clib.ctime(FileList[i].Create));
                 }
}
```

.fullpath(pathSpec)

This method converts `pathSpec` into an absolute path name. The parameter `pathSpec` must be a valid path specification. A string containing the full path specification is returned, corresponding to the conventions of an operating system. If `pathSpec` is not valid, the method returns `null`.

The following example returns the full specification of the current directory:.

```
function CurDir()
{
          return SElib.fullpath(".")
}
```

The following routine works in DOS or OS/2 to test whether a drive letter is valid.

```
function ValidDrive(DriveLetter)
{
          Clib.sprintf(CurdirSpec,"%c:.",DriveLetter)
          return (null != SElib.fullpath(CurdirSpec) )
}
```

.splitFilename(fileSpec)

This method Splits the parameter `fileSpec` into its various components which conform to the conventions of the operating system and returns an Object with the following properties.

`.dir` the directory name, including leading and drive separator characters

`.name` root name of the file

`.ext` extension name of the file, including preceding period

All properties are guaranteed to be non-`null`, and fileSpec can be reconstructed with the following statement.

```
var FileSpec = dir + name + ext;
```

The following lines are examples of using `SElib.splitFilename()`. The second example applies to a file structure for DOS and OS/2.

```
// sets parts.dir = "", parts.name = "foo", parts.ext = ""
var parts = SElib.splitFilename("foo")

// parts.dir = "..\\", parts.name = "*", parts.ext = ".doc"
parts = SElib.splitFilename("..\\*.doc")
```

Script execution

Scripts

.interpretInNewThread(filename, textToInterpret)

For Win32 and OS/2

This method creates a new thread within the current ScriptEase process and interprets a script within that new thread. The new script runs independently of the currently executing thread. This method differs from `.interpret()` in that the calling thread does not wait for the interpretation to finish and differs from `.spawn()` in that the new thread runs in the same memory and process space as the currently running thread. The method `.interpretInNewThread()` returns the ID number of the thread containing the new instance of ScriptEase. If there is an error, 0 or -1 is returned, depending on the operating system.

A script writer must ensure any synchronization among threads. ScriptEase data and globals are on a per-thread basis.

If parameter filename is not `null`, then it is the name of a file to interpret and the parameter `textToInterpret` is parsed as if having command-line parameters for a `main()` function. If filename is `null`, then `textToInterpret` will be treated as JavaScript code and interpreted directly.

This method is not supported on operating systems that do not support multithreading, such as DOS and 16-bit Windows.

.spawn(mode, ExecutableSpec [, arg1[, arg2[,...]]])

This method launches another application. The parameter mode determines the behavior of the script after the spawn call, while `ExecutableSpec` is the name of the process you are spawning. Any arguments to the spawned process follow.

The parameter mode may be one of the following values. Note that not all values are valid on all systems.

`P_WAIT`	Wait for a child program to complete before continuing. (All platforms)
`P_NOWAIT`	A script continues to run while a child program runs. In windows, a successful call with mode P_NOWAIT returns the window handle of the spawned process. (Windows and OS/2)
`P_SWAP`	Like P_WAIT, but swap out ScriptEase to create more room for child process. P_SWAP will free up as much memory as possible by swapping ScriptEase to EMS/XMS/INT15 memory or to disk (in TMP directory, else TEMP, else current directory) before executing the child process (thanks to Ralf Brown for his excellent spawn library). (DOS only)
`P_OVERLAY`	The script exits and child program is executed in its place. (DOS 16-bit)

If the parameter mode is `P_OVERLAY`, there is no return value. If mode is `P_WAIT`, the return is the exit code of the child process, else it is -1. If mode is `P_NOWAIT` or `P_SWAP`, the return is the identifier of the child process if successful, else it is -1.

The parameter `ExecutableSpec` may be the path and filename of an executable file or the name of a ScriptEase script. If it is a script, the spawned script runs from the same instance of ScriptEase as the calling script. A spawned script does not cause another instance of the interpreter to be launched. A script that has been bound with the ScriptEase /BIND function cannot be spawned from the same instance as the calling script.

The parameter `ExecutableSpec` is automatically passed as argument 0. ScriptEase implicitly converts the arguments into strings before passing them on to the child process.

The parameter `.spawn()` searches for `ExecutableSpec` in the current directory and then in the directories of the PATH environment variable. If there is no extension for ExecutableSpec, `.spawn()` searches first for .com, .exe files, .bat, and .cmd files, in this order.

If a batch file is being spawned in 16-bit DOS and the environment variable COMSPEC_ENV_SIZE exists, the command processor is provided the amount of memory as indicated by COMSPEC_ENV_SIZE. If COMSPEC_ENV_SIZE does not exist, the command processor receives only enough memory for existing environment variables.

A return value of -1 results in the setting of the property Clib.errno to identify why the function failed.

Example

The following example calls a mortgage program, Mortgage.exe, which takes three parameters, initial debt, rate, and monthly payment, and returns, in its exit code, the

number of months needed to pay the debt.

```
  var months = SElib.spawn(P_WAIT, "MORTGAGE.EXE 300000 10.5
1000");
  if (months < 0)
     Screen.writeln( "Error spawning MORTGAGE");
  else
     Clib.printf("It takes %d months to pay off the
                  mortgage\n", months);
```

The parameters could also be passed to Mortgage.exe as separate variables, as in the following.

```
  var months = SElib.spawn(P_WAIT,"MORTGAGE.EXE",300000,
                           10.5,1000);
```

The same arguments could be passed to Mortgage.exe in a variable array, provided that they are all of the same data type, in this case strings.

```
  var MortgageData;
  MortgageData[0] = "300000";
  MortgageData[1] = "10.5";
  MortgageData[2] = "1000";
  var ths = spawn(P_WAIT,"MORTGAGE.EXE",MortgageData);
```

See also the example for SElib.suspend().

.suspend(milliSecondDelay)

This method suspends program operation for the amount of time specified by `milliSecondDelay`.

`True` accuracy to the millisecond is not guaranteed, and is only approximated according to the accuracy provided by the underlying operating system. This method allows a computer to devote more time to other processes and can be used to give the processor time to complete tasks before calling the next line in a script.

The following example spawns a copy of Windows Notepad, puts the date and time into the document by simulating the selection of Time/Date from the Edit menu, and then displays the line "You asked for the time?". The `.suspend()` method gives the processor time to finish completing the menu command before entering the text into Notepad. If Keystroke() were called immediately after the call to MenuCommand(), the text would be sent to Notepad while the menu item was still being selected and would be garbled.

```
  #include "menuctrl.jsh"
  #include "keypush.jsh"
  var wHnd = SElib.spawn(P_NOWAIT, "notepad");
  MenuCommand(wHnd, "Edit##Time");
  SElib.suspend(300);
  Keystroke("\nYou asked for the time?");
```

In this example, execution is suspended for a little less than a third of a second. The delay is not noticeable to humans but gives a computer enough time to finish a task.

Windows

.baseWindowFunction(handle, message, param1, param2)

This method calls the base procedure of a window created with a WindowFunction in `.makeWindow()` or subclassed with `.subclassWindow()`. This method is normally used within a ScriptEase window function to pass the window parameter to the base procedure before handling it in your own code. Remember that if your window function returns no value, ScriptEase will call the base procedure automatically which is the preferred method.

This method returns the value returned by the base window function. If the parameter handle is not a window with a WindowFunction created with `.makeWindow()` or is not a window subclassed with `.subclassWindow()` then, the return is 0.

The parameter `handle` is a Window handle for the window receiving this message. This handle must be the window handle of a window created with `.makeWindow()` or subclassed with `.subclassWindow()`.

The parameter `message` is a message ID.

The parameter `param1` is the first parameter for this Message ID.

The parameter `param2` is the second parameter for this Message ID.

.breakWindow([WindowHandle])

For Win32 and Win16

This method releases control of a window controlled by `.subclassWindow()` or destroys a window previously created with `.makeWindow()`. No other windows are affected. `WindowHandle` is the handle of a window being destroyed or released. If it is not a valid window handle, no action is taken and `true` is returned.

When a window is destroyed all appropriate DestroyWindow() functions, internal to Windows itself, are called. Any child windows of the main window are destroyed before the main window.

If `WindowHandle` is a window controlled by `.subclassWindow()`, then this method removes the WindowFunction for a window from the message function loop.

If `WindowHandle` is not supplied, then all windows created with `makeWindow()` are destroyed and all subclassing ends.

If the method successfully destroys or subclasses its windows, the return is `true`, else the return is `false`.

.doWindows([boolean])

For Win32 and Win16.

This method interrupts standard script processing while allowing window functions to receive and process their messages. This is a way for your script to use as little CPU processing as is needed to handle windows it has created. Many of the flags that define window messages are kept in the library file, `Message.jsh`.

If the optional parameter boolean is `true`, `false` being the default, the method returns immediately, regardless of whether there were messages for this application or not. Otherwise this method yields control to other applications until a message has been processed, subject to filtering by `.messageFilter()`, for this application or for any window subclassed by this application.

This method returns `true` if any of the windows created with `.makeWindow()` or subclassed with `.subclassWindow()` are still open, that is, have not received WM_NCDESTROY. This method returns `false` if there remain no valid windows created by your script.

The following example displays a standard Windows window. If you click anywhere in the window, the string "You clicked me!" is displayed briefly. When the window is closed, the script terminates.

```
#include "message.jsh"
#include "window.jsh"
main()
{
          var hwnd = SElib.makeWindow(null, null,
              WindowFunction,
     "Display Windows' messages",
     WS_OVERLAPPEDWINDOW | WS_VISIBLE,
     CW_USEDEFAULT, CW_USEDEFAULT, 500, 350, null, 0);
     SElib.messageFilter(hwnd, WM_LBUTTONDOWN);
//execute until all my windows are closed
 while(SElib.doWindows());
}

function WindowFunction(hwnd, msg, parm1, parm2)
{
          if (msg == WM_LBUTTONDOWN)
          {
              var msgHwnd = SElib.makeWindow(hwnd, "static",
          null,
                                    "You clicked me!",
                              WS_CHILD | WS_VISIBLE,
                                    200, 150, 100, 50, null, 0);
                              SElib.suspend(1000);
                              SElib.breakWindow(msgHwnd);
          }
}
```

.makeWindow(Parent, Class, WindowFunction, Text, Style, Column, Row, Width, Height, CrParm[, UtilityVar])

For Win32 and Win16

This method makes a window for display and for receiving windows messages. Created windows will receive their messages during normal script processing (in the small interval between each ScriptEase statement, when a message is sent to that window, as opposed to posted messages), or during calls to `.doWindows`. The Window Class is registered if it is an unknown class, and WindowFunction is called by Windows for every Windows message.

This method is complex and forms the basis for all the behavior generally seen in a Windows program.

For the parameter `Parent`, if a window is to be created on the desktop, pass `null`. If it is a subwindow, pass the window handle of the main window.

The parameter `Class` may be either an object or a string. If it is a string, it must be the name of a pre-existing Windows' class, such as "edit" or "button." If it is an object, it may have the properties listed below. Properties that are not defined receive default values.

`.style`	style
`.icon`	bitmap icon for minimized window
`.cursor`	cursor appearance when over this window
`.background`	window background color

The parameter `WindowFunction` is a function that is called every time Windows sends a message to the new window. Set WindowFunction to `null` if no WindowFunction is to be called. If `WindowFunction` has a return value it must be a number. The function must have the following format.

```
int WindowFunction(Handle, Message, Param1, Param2, UtilityVar)
```

`Handle`	Window handle for window receiving this message
`Message`	The message ID
`Param1`	First parameter for this Message ID
`Param2`	Second parameter for this Message ID
`UtilityVar`	An optional variable used to pass data to the WindowFunction. See below.

Use the method `SElib.messageFilter()` to filter messages to the window. For more information about `WindowFunction`, see below.

The parameter `Text` is the window text or title. If there is no text, text is set to `null`, or an empty string.

The parameter `Style` is the window style, as defined by the `WS_xxxx` values in Windows.jsh.

The parameters `Column` and `Row` position the upper-left corner for the window when it is created. Use CW_USEDEFAULT to let Windows choose the position.

The parameters `Width` and `Height` set the dimensions of the window.

The parameter CrParm is generally set to `null`. It may be a number or object to pass with the window's WM_CREATE message.

The optional parameter `UtilityVar` is any variable that you choose to make available to the WindowFunction. This variable may be any kind of variable, including a structure if you want to pass many values to the `WindowFunction`. `WindowFunction` may also alter `UtilityVar`. If this parameter is not supplied, then no such variable will be available to the WindowFunction().

On success, the method returns WindowHandle of the created window. The method returns `null` for failure. See `.doWindows()` for an example.

.messageFilter(WinHandle[, Message1[, Message2[, . . .]]])

For Win32 and Win16

This method restricts the messages processed by ScriptEase created with `.makeWindow()` or subclassed with `.subclassWindow()`. Scripts run much faster if ScriptEase only processes the messages that they act on and lets the OS platform default processing on all other messages. Initially, there are no message filters so all messages are processed.

`WinHandle` is a window handle for a window created with `.makeWindow()` or subclassed with `.subclassWindow()`.

.multiTask(bool)

For Win16

Normally, multitasking is enabled and should be turned off only for very brief and critical sections of code. No messages are received by the current program or any other program while multitasking is off.

The method `.multiTask()` is additive, meaning that if you call `.multiTask(false)` twice, then you must call `.multiTask(true)` twice before multitasking is resumed.

The following section of code empties the clipboard. Multitasking is turned off during this brief interval to ensure that no other program tries to open the clipboard while this program is accessing it.

```
SElib.multiTask(false);
SElib.dynamicLink("USER", "OPENCLIPBOARD", SWORD16, PASCAL,
                  Screen.handle());

SElib.dynamicLink("USER", "EMPTYCLIPBOARD", SWORD16, PASCAL);
SElib.dynamicLink("USER", "CLOSECLIPBOARD", SWORD16, PASCAL);
SElib.multiTask(true);
```

.subclassWindow(WindowHandle, WindowFunction[,UtilityVar])

For Win32 and Win16

This method hooks the specified `WindowFunction` into the message loop for a window such that the function is called before the window's default or previously-defined function. This method is most powerful when used to modify the behavior of windows belonging to other processes.

The parameter `WindowHandle` is the window handle of an already existing window to subclass.

The parameter `WindowFunction` is the same as in the `.makeWindow()` method. Note that, as in the `.makeWindow()` method, if this method returns a value, then the default or subclassed function is not called. If this method returns no value, then call is passed on to the previous function. This method may be used to subclass any Window that is not already being managed by a WindowFunction for this ScriptEase instance. If a window was created with `.makeWindow()` or is already subclassed then this method fails.

A WindowFunction may modify UtilityVar.

In your function that handles messages for another process, certain limits are set as to what you can do with system resources. For example, an open file handle is invalid while processing a message for another program, because Windows maps any file handles into a table for that other program. To work around this problem, you may want to send a message to one of your ScriptEase windows to handle the processing. This action switches Windows' tables to your program while handling that SendMessage.

This method returns `false` if WindowHandle is invalid, was created with `.makeWindow()`, or is already subclassed, else it returns `true`.

.windowList([WinHandle])

For Win32 and Win16

This method returns an array of all child window handles of `WinHandle`. If `WinHandle` is not supplied, it will return an array of all windows on the desktop.

Dynamic links

For Win32, Win16, and OS/2

This method allows flexibility when making calls to dynamic link libraries, DLLs, and allows access to operating-system functions not explicitly provided by ScriptEase. If you know the proper conventions for a call, then you can make a `.dynamicLink()` call in a ScriptEase function to be used for making a system call. Such a function is referred to as a wrapper, a function in which a system call becomes available as a function call.

There are three versions of `.dynamicLink()`: Win32, Win16, and OS/2. These three versions differ slightly in the way they are called. So, if you wish to use one function in a script that will be run on different platforms, you must create an operating system filter using preprocessor directives: `#if`, `#ifdef`, `#elif`, `#else`, and `#endif`.

Since the three versions are different in the way that they call .dynamicLink(), they will be treated separately.

.dynamicLink(Library, Procedure, Convention, . . .)

For Win32

The parameter `Library` is the name of the dynamic link library in which the procedure is located.

The parameter `Procedure` is the name or ordinal number of the procedure in the `Library` dynamic link library.

The parameter `Convention` specifies the calling convention and may be one of the following.

`CDECL`	Push right parameter first; Caller pops parameters.
`STDCALL`	Push right parameter first; Caller pops parameters. This is almost always the option used in Win32.
`PASCAL`	Push left parameter first; Callee pops parameters.

All values are passed as 32-bit values. If a parameter is undefined when `dynamicLink()` is called, then it is assumed that the parameter is a 32-bit value to be filled in, that is, the address of a 32-bit data element is passed to the function, and that function will set the value.

If any parameter is a structure then it must be a structure that defines the binary data types in memory to represent the following variable. Before calling the DLL function, the structure is copied to a binary buffer as described in `Blob.put()` and `Clib.fwrite()`. After calling the DLL function, the binary data will be converted back into the data structure according to the rules defined in `Blob.get()` and `Clib.fread()`. Data conversion is performed according to the current `_BigEndianMode` setting.

The following routine calls the Windows MessageBeep() function:

```
SElib.dynamicLink("USER32","MessageBeep",STDCALL,0);
```

The following example displays a simple message box and waits for user to press `Enter`.

```
#include <msgbox.jsh>

MessageBox("The following samples show various ways\n"
           "to use the Windows MessageBox() function.\n"
           "A wrapper for MessageBox() is located in\n"
           "the jse library: \"MsgBox.jsh\"",
           "MsgBoxes - Welcome!");

MessageBox("This example only passes 1 parameter: the message
            string.");

MessageBox("This passes the message string and a
                         title.","Here is the title.");

MessageBox("This example passes a zero-length string to get
            no title.","");

switch ( MessageBox("You can also offer different
                  choices.\nPick one now...", "MessageBox()
                choices",
         MB_YESNOCANCEL) ) {
   case IDYES:    button = "Yes";   break;
   case IDNO:     button = "No";    break;
   case IDCANCEL: button = "Cancel";   break;
}

Clib.sprintf(message,"You selected the %s button",button);
MessageBox(message,"");
```

.dynamicLink(Library, Procedure, RetType, Convention, . . .)

For Win16

The parameter `Library` is the name of the dynamic link library in which the procedure is located.

The parameter `Procedure` is the name or ordinal number of the procedure in the `Library` dynamic link library.

The parameter `RetType` tells ScriptEase what type of value the procedure returns, so that it can be properly converted into a ScriptEase number. Return types, such as SWORD16 and UWORD32, are described in the `Clib.fread()` function.

The parameter Convention specifies the calling Convention and may be one of the following.

`CDECL`	Push right parameter first; Caller pops parameters.
`PASCAL`	Push left parameter first; Callee pops parameters.

If the parameter is a `BLOb`, a byte-array, or an undefined value, it is passed as a 32-bit far pointer. All other numeric values are passed as 16-bit values; if 32-bits are needed for an

integer, the parameter must be passed in parts, with the low word first and the high word second for CDECL calls but the high word first and low word second for PASCAL calls.

If a parameter is undefined when `.dynamicLink()` is called, then it is assumed that the parameter is a far pointer to be filled in, that is, that the far address of an integer data element is passed to the function and that function will set the value. If any parameter is a structure, then it must be a structure that defines the binary data types in memory to represent the following variable. Before calling the DLL function, the structure will be copied to a binary buffer as described in `Blob.put()` and `Clib.fwrite()`. After calling the DLL function, the binary data is converted back into the data structure according to the rules defined in `Blob.get()` and `Clib.fread()`. Data conversion is performed according to the current `_BigEndianMode` setting.

.dynamicLink(Library, Ordinal, BitSize, Convention, . . .)

For OS/2

The parameter `Library` is the name of the dynamic link library in which the procedure is located.

The parameter `Ordinal` is the name or ordinal number of the procedure in the `Library` dynamic link library.

The parameter `BitSize` defines whether this is a 16-bit or a 32-bit call. It may be either one of the pre-defined values: `BIT16` or `BIT32`.

Calling Convention may be any of the following:

`CDECL`	Push right parameter first; Caller pops parameters.
`STDCALL`	Push right parameter first; Caller pops parameters.
`PASCAL`	Push left parameter first; Callee pops parameters.

The OS/2 processor also allows you to call a function via a call gate with the following syntax:

```
SElib.dynamicLink(CallGate, BitSize, Convention, . . . )
```

where `CallGate` is the gate selector for a routine referenced through a call gate.

Any parameters required by a dynamically linked function should be passed at the end of the parameters listed above. These variables are interpreted as follows, depending on the operating system.

For 32-bit functions, all values are passed as 32-bit values. For 16-bit functions, if the parameter is a BLOB, a byte-array, or an undefined value, then it is passed as a 16:16 segment:offset pointer, otherwise all numeric values are passed as 16-bit values, so if 32-bits are needed they must be passed in parts, with the low word first and the high word second.

If a parameter is undefined when .dynamicLink() is called, then it is assumed that parameter is a 32-bit value to be filled in, that is, that the address of a 32-bit data element is passed to the function and that function will set the value. If any parameter is a structure then it must be a structure that defines the binary data types in memory to represent the following variable. Before calling the DLL function, the structure is copied

to a binary buffer as described in `Blob.put()` and `Clib.fwrite()`. After calling the DLL function, the binary data is converted back into the data structure according to the rules defined in `Blob.get()` and `Clib.fread()`. Data conversion is performed according to the current `_BigEndianMode` setting. Any of these calls return the value returned by the dynamically-linked function as interpreted according to `RetType`.

General

.getObjectProperties(ObjectVar[, IncludeUndefinedProperties])

This method returns an array of strings and each string is the name of a property of the Object passed as `ObjectVar`.

The parameter `IncludeUndefinedProperties` must be `true` to return properties that are not defined. If `IncludeUndefinedProperties` is `false`, then only properties that have defined data are included. The default for `IncludeUndefinedProperties` is `false`.

The final member of the returned array returned is always `null`. If `ObjectVar` is not defined or contains no properties to be displayed, then the return is an array with a single element set to `null`.

The following example:

```
var Point;
Point.row = 5;
Point.col = 8;
Point.height;
PrintAllStructureMembers(Point);

function PrintAllStructureMembers(ObjectVar)
{
          Screen.writeln("Object Properties:");
          var MemberList = SElib.getObjectProperties(ObjectVar);
          for (var i = 0; null != MemberList[i]; i++)
  Screen.writeln("  " + MemberList[i]);
}
```

produces the following output.

```
Object Properties:
  row
  col
```

.inSecurity(infoVar)

This method calls the security manager's initialization routine and is the only way your application can directly interact with the security filter. It is provided so you can reinitialize the security system, probably to change the security level of a script.

Typically, you use this method when executing a particularly insecure piece of code, such as a script received over a network, to downgrade the security level, restoring it when the script completes.

The parameter infoVar is the variable to be passed to the security filter. Your application and its security filter may use it however you choose.

The return is `true` if there is a security filter, and `false` if there is not.

The Clib object

The `Clib` object contains functions that are a part of the standard C library. Methods to access files, strings, and characters are all part of the `Clib` object. Some functions have been moved to the section on redundant methods at the end of this descriptions of the `Clib` object. These methods may be considered redundant since their actions have been duplicated by JavaScript objects. Redundant functions may be used if desired since they still work fully.

Console I/O functions

.printf(formatString, [variables...])

This method writes output to the standard output device according to the format string and returns a number equal to the number of characters written, or a negative number if there is an error. The format string can contain character combinations indicating how following parameters are to be treated. Characters are printed as read to standard output until a percent character, %, is reached. % indicates that a value is to be printed from the parameters following the format string. Each subsequent parameter specification takes from the next parameter in the list following format. A parameter specification takes this form (square brackets indicate optional fields, angled brackets indicate required fields):

```
%[flags][width][.precision]<type>
```

flags may be:

`-`	Left justification in the field with blank padding; else right justifies with zero or blank padding
`+`	Force numbers to begin with a plus (+) or minus (-)
`blank`	Negative values begin with a minus (-); positive values begin with a blank
`#`	Convert using the following alternate form, depending on output data type:
`c,s,d,i,u`	No effect
`o`	0 (zero) is prepended to non-zero output
`x,X`	0x, or 0X, are prepended to output
`f,e,E`	Output includes decimal even if no digits follow decimal
`g,G`	Same as e or E but trailing zeros are not removed

width may be:

`n`	(n is a number e.g., 14) At least n characters are output, padded with blanks
`0n`	At least n characters are output, padded on the left with zeros
`*`	The next value in the argument list is an integer specifying the output width
`.precision`	If precision is specified, then it must begin with a period (.), and may be as follows:
`.0`	For floating point type, no decimal point is output
`.n`	n characters or n decimal places (floating point) are output
`.*`	The next value in the argument list is an integer specifying the precision width

type may be:

`d,i`	signed integer
`u`	unsigned integer
`o`	octal integer x
`x`	hexadecimal integer with 0-9 and a, b, c, d, e, f
`X`	hexadecimal integer with 0-9 and A, B, C, D, E, F
`f`	floating point of the form [-]dddd.dddd
`e`	floating point of the form [-]d.ddde+dd or [-]d.ddde-dd
`E`	floating point of the form [-]d.dddE+dd or [-]d.dddE-dd
`g`	floating point of f or e type, depending on precision
`G`	floating point of For E type, depending on precision
`c`	character ('a', 'b', '8', e.g.)
`s`	string

To include the % character as a character in the format string, you must use two % characters together, `%%`, to prevent the computer from trying to interpret it as one of the above forms.

Example:

Each of the following lines shows a printf example followed by what would show on the output in boldface:

```
Clib.printf("Hello world!")
     Hello world!
Clib.printf("I count: %d %d %d.",1,2,3)
     I count: 1 2 3
var a = 1;
var b = 2;
Clib.printf("%d %d %d", a, b, a +b)
     1 2 3
```

.getch()

This method works exactly like getche(), but does not echo the returned key to the screen. For example, the following code has you enter a password; each time you enter a letter an asterisk is written to the screen:

```
var password;
for (var gg = 0; ;gg++)
{
          var letter = Clib.getch();
          if (letter == '\n') continue;
          Clib.putchar( '*'.charCodeAt(0) );
          password[gg] = letter;
}
```

.getchar()

This method returns the next character from stdin. Usually, this is the keyboard, but you may redefine it to something else. This method will wait for "enter" to be pressed after the key, and will then return two values: the key pressed, and then the value of the enter key.

.getche()

This method waits until a key is pressed and returns the character value of that key. The character will be printed (echoed) to the screen. Some key presses, such as extended keys and function keys, may generate multiple getche() return values. If a key was pressed before calling the function but never cleared from the keyboard buffer, that value will be returned instead of the next pressed key. This is not a common occurrence but can happen. To see whether there are any key values pending in the keyboard buffer, use `.kbhit()`.

.gets()

This method reads an entire string from the keyboard and returns it (or `null` if there was an error). The function will read all characters up to a newline character or `EOF`. If a newline character is read, it will not be included in the string.

.kbhit()

This method checks to see whether there are any keystrokes waiting to be processed, returning `true` if there are and `false` if there are not.

.putchar(c)

This method writes the character c to the stream defined by stdout (usually the screen). If successful, it will return the character it just wrote; if not, it will return `EOF`.

This method is identical to `.fputc(c, stdout)`.

.puts(string)

Writes the string to stdout, followed by a newline character. It will not write the final `null` character of `null`-terminated strings. It returns EOF if there is an error writing the string; otherwise it returns a positive number.

This method is identical to `.fputs(s, stdout)` except that a newline character is written after the string.

.scanf()

This flexible method reads input from the screen, extracts data from it by matching the string to a format string (as described below), and stores the data in the variables which follow the format string. It returns the number of input items assigned; this number may be fewer than the number of parameters requested if there was a matching failure. The format string contains character combinations that specify the type of data expected. The format string specifies the admissible input sequences, and how the input is to be converted to be assigned to the variable number of arguments passed to this function. Characters are matched against the input as read and as it matches a portion of the format string until a `%` character is reached. `%` indicates that a value is to be read and stored to subsequent parameters following the format string. Each subsequent parameter after the format string gets the next parsed value taken from the next parameter in the list following format. A parameter specification takes this form (square brackets indicate optional fields, angled brackets indicate required fields):

```
%[*][width]<type>
```

***, width, and type may be:**

*	Suppress assigning this value to any parameter	
width	maximum number of characters to read; fewer will be read if whitespace or nonconvertible character	
type	may be one of the following:	
	`d,D,i,I`	signed integer
	`u,U`	unsigned integer
	`o,O`	octal integer
	`x,X`	hexadecimal integer
	`f,e,E,g,G`	floating point number
	`c`	character; if width was specified then this will be an array of characters of the specified length
	`s`	string
	`[abc]`	string consisting of all characters within brackets; where A-Z represents range "A" to "Z"
	`[^abc]`	string consisting of all character NOT within brackets.

Modifies any number of parameters following the format string, setting the parameters to data according to the specifications of the format string.

.vprintf(stream, valist)

This method displays formatted output on the standard output stream, screen, using a variable number of arguments. This method is similar to `.printf()` except that it takes a variable argument list using `valist`. See `.printf()` and `.va_start()` for more information. The method `.vprintf()` returns the number of characters written on success, else a negative number on error.

The following function acts just like a `.printf()` statement except that it beeps, displays a message, beeps again, and waits a second before returning. This method could be a wrapper for the `.printf()` method to display urgent messages.

```
function UrgentPrintf(FormatString /*, arg1,arg2, . . . */)
{
         //create variable arg list
         Clib.va_start(va_list, FormatString);
         Screen.write("\a");   // audible beep
         // printf original statement
         var ret = Clib.vprintf(FormatString, va_list);
         Screen.write("\a");   // beep again
         SElib.suspend(1000);  // wait a second before returning
         Clib.va_end(va_list); // end using va_list
         return(ret);          // return as printf would }
}
```

.vscanf(formatstring, valist)

This method gets formatted input from the standard input stream, the keyboard, using a variable number of arguments. This method is similar to `.scanf()` except that it takes a variable argument list. See `.scanf()` and `.va_start()` for more information.

The method `.vscanf()` modifies any number of parameters following formatstring, setting the parameters to data according to the specifications of the format string. These parameters are specified by `valist`.

This method returns the number of input items assigned. This number may be fewer than the number of parameters requested if there is a matching failure during input.

The following function behaves like `.scanf()`, including taking a variable number of input arguments, except that it beeps and tries again if there are zero matches.

```
function Must_scanf(FormatString /*, arg1,arg2,arg3, . . .*/)
   {
     Clib.va_start(va_list, FormatString);
     // creates variable arg list
     do
     { // mimic original scanf() call
       var count = Clib.vscanf(FormatString, va_list);
       if ( 0 == count ) // if no match, then beep
       Screen.write("\a");
     } while( 0 == count );
       // if not match, then try again
       Clib.va_end(va_list);
       // end using va_list (optional)
       return(count);
       // return as scanf would
   }
```

Time functions

The `Clib` object (like the `Date` object) represents time in two distinct ways: as an integral value (the number of seconds passed since January 1, 1970) and as a `Time` object with properties for the day, month, year, etc. This `Time` object is distinct from the standard JavaScript Date object. You cannot use `Date` object properties with a `Time` object or vice versa.

In the methods below, `Time` represents a variable in the `Time` object format, while `timeInt` represents an integral time value.

.asctime(Time)

Returns a string representing the date and time extracted from a `Time` object (as returned by `.localtime()`). The string will have this format:

```
Mon Jul 19 09:14:22 1993
```

.clock()

Returns the current processor tick count. Clock value starts at 0 when ScriptEase program begins and is incremented CLOCKS_PER_SEC times per second.

.ctime(timeInt)

This method is equivalent to: `Clib.asctime(Clib.localtime(time))`, where `timeInt` is a date-time value as returned by the `.time()` function.

.difftime(timeInt0, timeInt1)

This method returns the difference in seconds between two times. `timeInt0` and `timeInt1` are integral time values as returned by the `.time()` function.

.gmtime(timeInt)

Takes the integer `timeInt` (as returned by the `time()` function) and converts it to a `Time` object representing the current date and time expressed as Greenwich mean time. See `localtime()` for a description of the returned object.

.localtime(timeInt)

This method returns the value `timeInt` (as returned by the `time()` function) as a `Time` object. Note that the `Time` object differs from the `Date` object, although they contain similar data. The `Time` object is for use with the other date and time functions in the `Clib` object. It has the following integer properties:

`.tm_sec`	second after the minute (from 0)
`.tm_min`	minutes after the hour (from 0)
`.tm_hour`	hour of the day (from 0)
`.tm_mday`	day of the month (from 1)
`.tm_mon`	month of the year (from 0)
`.tm_year`	years since 1900 (from 0)
`.tm_wday`	days since Sunday (from 0)
`.tm_yday`	day of the year (from 0)
`.tm_isdst`	daylight-savings-time flag

The following function prints the current date and time on the screen and returns the day of the year, where Jan 1 is the 1st day of the year:

```
ShowToday()
// show today's date; return day of the year in USA format
{
          // get current time structure
          var tm = localtime(time());
          // display the date in USA format
          Clib.printf("Date: %02d/%02d/%02d", tm.tm_mon+1,
                      tm.tm_mday, tm.tm_year % 100);
          // convert hour to run from 12 to 11, not 0 to 23
          var hour = tm.tm_hour % 12;
          if ( hour == 0 )
              // print current time
              Clib.printf("Time: % 2d:%02d:%02d\n", hour,
          tm.tm_min, tm.tm_sec);
              // return day of year, where Jan. 1 would be day 1
          return( tm.tm_yday + 1 );
}
```

.mktime(Time)

This method converts `Time` (an object as returned by `.localtime()`) to the time format returned by `.time()` (an integer). All undefined elements of `Time` will be set to 0 before the conversion. It returns -1 if time cannot be converted or represented.

In other words, while `.localtime()` converts from a time integer to a `Time` object, `.mktime()` converts from a `Time` object to a time integer.

.strftime(&stringVar, formatString, Time)

This method creates a string that describes the date and or time and stores it in the variable stringVar. formatString describes what the string will look like; Time is a time object as returned by .localtime().

These following conversion characters are used with strftime() to indicate time and date output:

%a	abbreviated weekday name (Sun)
%A	full weekday name (Sunday)
%b	abbreviated month name (Dec)
%B	full month name (December)
%c	date and time (Dec 2 06:55:15 1979)
%d	two-digit day of the month (02)
%H	two-digit hour of the 24-hour day (06)
%I	two-digit hour of the 12-hour day (06)
%j	three-digit day of the year from 001 (335)
%m	two-digit month of the year from 01 (12)
%M	two-digit minute of the hour (55)
%p	AM or PM (AM)
%S	two-digit seconds of the minute (15)
%U	two-digit week of the year where Sunday is first day of the week (48)
%w	day of the week where Sunday is 0 (0)
%W	two-digit week of the year where Monday is first day of the week (47)
%x	the date (Dec 2 1979)
%X	the time (06:55:15)
%y	two-digit year of the century (79)
%Y	the year (1979)
%Z	name of the time zone, if known (EST)
%%	the per cent character (%)

Example:

The following code displays the full day name and month name of the current day:

```
Clib.strftime(TimeBuf,"Today is: %A, and the month is: %B",
           Clib.localtime(time()));
Clib.puts(TimeBuf);
```

.time([&t])

Returns an integer representation of the current time. The format of the time is not specifically defined except that it represents the current time, to the system's best approximation, and can be used in many other time-related functions. If t is supplied then it will be set to equal the returned value.

Script execution

.abort([AbortAll)

This method terminates a program, usually when a specified error occurs. This method causes abnormal program termination and should only be called on a fatal error. This method exits, without returning to the caller, and returns EXIT_FAILURE to the operating system.

If the boolean `AbortAll` is `true`, this method aborts through all levels of ScriptEase interpretation. If you are in multiple levels of `.interpret()`, `.abort(true)` aborts through all `.interpret()` levels.

.assert(boolean)

If boolean evaluates to `false` this function will print the file name and line number to `stderr` and abort. If the assertion evaluates to `true` then the program continues. `.assert()` is typically used as a debugging technique to test assumptions before executing code based on those assumptions. Unlike C, the ScriptEase implementation of assert does not depend upon NDEBUG being defined or undefined; it is always active.

The Inverse() function below returns the inverse of the input number (i.e., 1/x):

```
function Inverse(x) // return 1/x
{
          assert(0 != x);
          return 1 / x;
}
```

.atexit(exit Function)

This method registers a function to be called when the script ends. The variable `exit function` passed to this function is a function to be called.

.exit(status)

This method causes normal program termination. It calls all functions registered with `.atexit()`, flushes and closes all open file streams, updates environment variables if applicable to this version of ScriptEase, and returns control to the OS environment with the return code of status.

.system(commandString)

.system(P_SWAP, commandString) (DOS versions only)

Passes `commandString` to the command processor and returns whatever value was returned by the command processor. `commandString` may be a formatted string followed by variables according to the rules defined in `.sprintf()`.

DOS	In the DOS version of ScriptEase, if the special argument P_SWAP is used then SeDos.exe is swapped to EMS/XMS/INT15 memory or disk while the system command is executed. This leaves almost all available memory for executing the command. See SElib.spawn() for a discussion of P_SWAP.
DOS32	The 32-bit protected mode version of DOS ignores the first parameter if it is an not a string

; in other words, P_SWAP is ignored.

Error

.errno

The property .errno stores diagnostic message information when a function fails to execute correctly. Many functions in the `Clib` and `SElib` objects set .errno to non-zero in case of error to provide more specific information about the error. ScriptEase implements `.errno` as a macro to the internal function `_errno()`. This property can be accessed with `.perror()` or `.strerror()`.

.clearerr(filePointer)

This method clears the error status and resents the end-of-file flags for the file associated with `filePointer`. There is no return value.

.ferror(filePointer)

The parameter filePointer is a file pointer as returned by `.fopen()`. This method tests and returns the error indicator for stream file. Returns 0 if no error, otherwise returns the error value.

.perror([&string])

Prints and returns an error message that describes the error defined by `.errno`. This method is identical to calling .strerror(errno). If a string variable is supplied it will be set to the string returned.

.strerror(errno)

When some functions fail to execute properly, they store a number in the .errno property. The number corresponds to the type of error encountered. This method converts the error number to a descriptive string and returns it.

This method opens a file for reading, and if it cannot open the file then it prints a descriptive message and exits the program:

```
function MustOpen(filename)
{
          var fh = fops(filename,"r");
          if ( fh == null )
          {
              Clib.printf("Error:%s\n",strerror(errno));
              Clib.exit(EXIT_FAILURE);
          }
          return(fh);
}
```

File I/O:

.fopen(filename, mode)

This method opens the file specified by `filename` for file operations specified by `mode`, returning a file pointer (filePointer) to the file opened. `null` is returned in case of failure.

The parameter `filename` is a string. It may be any valid file name, excluding wildcard characters.

The parameter mode is a string composed of "r", "w", or "a" followed by other characters as follows:

r	open file for reading; file must already exist
w	open file for writing; create if doesn't exist; if file exists then truncate to zero length
a	open file for append; create if doesn't exist; set for writing at end-of-file
b	binary mode; if b is not specified then open file in text mode (end-of-line translation)
t	text mode
+	open for update (reading and writing)

When a file is successfully opened, its error status is cleared and a buffer is initialized for automatic buffering of reads and writes to the file.

The following code opens the text file "ReadMe" for text-mode reading, and displays each line in that file:

```
var fp = Clib.fopen("ReadMe","rt");
if ( fp == null )
          Clib.printf("\aError opening file for reading.\n")
else
          while ( null != (line=Clib.fgets(fp)) )
          {
              Clib.fputs(line, stdout)
          }
          Clib.fclose(fp);
```

.fclose(filePointer)

`filePointer` is a file pointer as returned by `.fopen()`. This method flushes the stream's file buffers and closes the file. The file pointer ceases to be valid after this call. Returns zero if successful, otherwise returns EOF.

.feof(filePointer)

`filePointer` is a file pointer as returned by `.fopen()`. This method returns an integer which is non-zero if the file cursor is at the end of the file, and 0 if it is NOT at the end of the file.

.fflush(filePointer)

Causes any unwritten buffered data to be written to `filePointer`. If `filePointer` is `null` then flushes buffers in all open files. Returns zero if successful; otherwise EOF.

.fgetc(filePointer)

This method returns the next character in the file stream indicated by `filePointer` as a byte converted to an integer. If there is a read error or the file cursor is at the end of the file EOF will be returned. If there is a read error then `ferror`() will indicate the error condition.

.fgetpos(filePointer, &pos)

This method stores the current position of the file stream `filePointer` for future restoration using `.fsetpos()`. The file position will be stored in the variable pos; use it with .`fsetpos`() to restore the cursor to its position. Returns zero for success, otherwise returns non-zero and stores an error value in .`errno`.

.fgets([number,] filePointer)

This method returns a string consisting of the characters in a file from the current file cursor to the next newline character. The newline will be returned as part of the string. If there is an error or the end of the file is reached `null` will be returned.

A second syntax of this function takes a number as its first parameter. This number is the maximum length of the string to be returned if no newline character was encountered.

.fprintf(filePointer, formatString,...)

This flexible function writes a formatted string to the file associated with `filePointer`. The second parameter, `formatString`, is a string of the same pattern as `.sprintf`() and `.rsprintf`.

.fputc(charVar, filePointer)

If `charVar` is a string, the first character of the string will be written to the file indicated by `filePointer`. If `charVar` is a number, the character corresponding to its unicode value will be added.

If successful, the character written will be returned, otherwise EOF is returned.

.fputs(string, filePointer)

This method writes the value of string to the file indicated by `filePointer`. Returns EOF if write error, else returns a non-negative value.

.fread(&destVar, varDescription, fp)

This method reads data from an open file and stores it in `destVar`. If it does not yet exist `destVar` will be created. `varDescription` is a variable that describes the how and how much data is to be read: if `destVar` is a buffer, it will be the length of the buffer; if `destVar` is an object, `varDescription` must be an object descriptor; and if `destVar` is to hold a single datum then `varDescription` must be one of the following.

`UWORD8`	Stored as a byte in DestVar
`SWORD8`	Stored as an integer in DestVar
`UWORD16`	Stored as an integer in DestVar
`SWORD16`	Stored as an integer in DestVar
`UWORD24`	Stored as an integer in DestVar
`SWORD24`	Stored as an integer in DestVar
`UWORD32`	Stored as an integer in DestVar
`SWORD32`	Stored as an integer in DestVar
`FLOAT32`	Stored as a float in DestVar
`FLOAT64`	Stored as a float in DestVar
`FLOAT80`	Stored as a float in DestVar (not available in some systems)

In all cases, this function returns the number of elements read. For DestBuffer this would be the number of bytes read, up to bufferLen. For DataTypeInFile this returns 1 if the data is read or 0 if read error or end-of-file is encountered.

For example, the definition of a structure might be:

```
ClientDef = new blobDescriptor();
ClientDef.Sex = UWORD8;
ClientDef.MaritalStatus = UWORD8;
ClientDef._Unused1 = UWORD16;
ClientDef.FirstName = 30; ClientDef.LastName = 40;
ClientDef.Initial = UWORD8;
```

The ScriptEase version of `fread`() differs from the standard C version in that the standard C library is set up for reading arrays of numeric values or structures into consecutive bytes in memory. In JavaScript this is not necessarily the case.

Data types will be read from the file in a byte-order described by the current value of the `_BigEndianMode` global variable.

To read the 16-bit integer "`i`", the 32-bit float "`f`", and then 10-byte buffer "`buf`" from the open file "`fp`" use code like the following.

```
if ( !Clib.fread(i,SWORD16,fp) || !Clib.fread(f,FLOAT32,fp)
                               || 10 != Clib.fread(buf,10,fp) )
{
          Clib.printf("Error reading from file.\n");
          Clib.abort();
}
```

.freopen(filename, mode, oldFp)

This method closes the file associated with `oldFp` (ignoring any close errors), and then opens filename according to `mode` (as in .`fopen`()) , and reassociates `oldFp` to this new file specification. This method is commonly used to redirect one of the pre-defined file handles (`stdout`, `stderr`, `stdin`) to or from a file.

The method returns a copy of the modified `oldFp`, or `null` if it fails.

This sample code will call the ScriptEase for DOS program with no parameters (which causes a help screen to be printed), but redirecting stdout to a file se.out so that se.out will contain the text of the ScriptEase help screens.

```
if ( null == Clib.freopen("SE.OUT","w",stdout) )
          Clib.printf("Error redirecting stdout\a\n")
else
          Clib.system("SEDOS.EXE");
```

.fscanf(filePointer, formatString [, ...])

This flexible function reads input from the file indicated by `filePointer` and stores in parameters following formatString according the character combinations in the format string, which indicate how the file data is to be read and stored. The file must be open, with read access. `.fscanf` returns the number of input items assigned. This number may be fewer than the number of parameters requested if there was a matching failure. If there is an input failure (before the conversion occurs) this function returns EOF.

See .`scanf`() for a description of this format string. The parameters following the format string will be set to data according to the specifications of the format string.

Given the following text file, Weight.dat:

```
Crow, Barney      180
Claus, Santa      306
Mouse, Mickey     2
```

the following code:

```
var fp = Clib.fopen("WEIGHT.DAT","r");
var FormatString = "%[,] %*c %s %d\n";
while (3 == Clib.fscanf(fp, FormatString, LastName, Firstame,
                        weight))
            Clib.printf("%s %s weighs %d pounds.\n",
               FirstName, LastName, weight);
Clib.fclose(fp);
```

results in the following output:

```
Barney Crow weighs 180 pounds.
Santa Claus weighs 306 pounds.
Mickey Mouse weighs 2 pounds.
```

.fseek(filePointer, offset [,int mode])

Set the position of the file pointer of the open file stream `filePointer`. The parameter `offset` is a number indicating how many bytes the new position will be past the starting point indicated by `mode`.

The parameter mode can be any of the following predefined values.

`SEEK_CUR`	seek is relative to the current position of the file
`SEEK_END`	position is relative from the end of the file
`SEEK_SET`	position is relative to the beginning of the file

If mode is not supplied then absolute offset from the beginning of file (SEEK_SET) is assumed. For text files (i.e., not opened in binary mode) the file position may not correspond exactly to the byte offset in the file.

This method returns zero for success, or non-zero if it cannot set the file pointer to the indicated position.

.fsetpos(filePointer, pos)

This method sets the current file stream pointer to the value defined by `pos`, which must be a value obtained from a previous call to `.fgetpos()` on the same open file. Returns zero for success, otherwise returns non-zero and stores an error value in errno.

.ftell(filePointer)

This method sets the position offset of the file pointer of an open file stream from the beginning of the file. For text files (i.e., not opened in binary mode) the file position may not correspond exactly to the byte offset in the file. Returns the current value of the file position indicator, or -1 if there is an error, in which case an error value will be stored in `errno`.

.fwrite(sourceVar, varDescription, filePointer)

This method writes the data in `sourceVar` to the file indicated by `filePointer` and returns the number of elements written. 0 will be returned if a write error occurs; use `.ferror()` to get more information about the error. `varDescription` is a variable that describes the how and how much data is to be read: if sourceVar is a buffer, `varDescription` will be the length of the buffer; if `sourceVar` is an object, `varDescription` must be an object descriptor; and if `sourceVar` is to hold a single datum then `varDescription` must be one of the values listed in the description for `.fread`.

The ScriptEase version of `fwrite()` differs from the standard C version in that the standard C library is set up for writing arrays of numeric values or structures from consecutive bytes in memory. This is not necessarily the case in JavaScript.

To write the 16-bit integer "i", the 32-bit float "f", and then 10-byte buffer "buf" into open file "fp" use the following code.

```
  if ( !Clib.fwrite(i, SWORD16, fp) || !Clib.fwrite(f, FLOAT32,
fp)
                            || 10 !=  fwrite(buf, 10, fp))
  {
            Clib.printf("Error writing to file.\n");
            Clib.abort();
  }
```

.getc(filePointer)

This method is identical to `.fgetc()`. It returns the next character in the file `filePointer` as a byte (unsigned) converted to an integer. Returns EOF if there is a read error or if at end-of-file; if there is a read error then ferror() will indicate the error condition.

.putc(char, stream)

This method writes the character `char`, converted to a byte, to the output file stream. This method is identical to `.fputc(c, stream)`. It returns `char` on success and EOF on a write error.

.remove(filename)

This method deletes the file specified by `filename`. Returns zero if successful and non-zero for failure.

.rename(oldFilename, newFilename)

This method renames `oldFilename` to `newFilename`. Both `oldFilename` and `newFilename` are strings. Returns zero if successful and non-zero for failure.

.rewind(fp)

This method sets the file cursor to the beginning of file. This call is identical to `.fseek(stream, 0, SEEK_SET)` except that it also clears the error indicator for this stream.

.tmpfile()

This method returns the file variable of a temporary binary file that will automatically be removed when it is closed or when the program exits. `null` will be returned if the function fails.

.tmpnam([&string])

This method creates a string that is a valid file name that is not the same as the name of any existing file and not the same as any filename returned by this function during execution of this program. If a string is supplied it will be set to the string that will be returned by this function.

.ungetc(char, stream)

This method pushes a character `char` back into an input `stream`. When `char` is put back it is converted to a byte and is again in an input stream for subsequent retrieval. Only one character is guaranteed to be pushed back. The method returns `char` on success, else EOF on failure.

Directory

.chdir(dirpath)

This method changes the directory for a script from its current directory to the directory specified in the parameter `dirpath`. The specified directory may be an absolute or relative path specification.

.getcwd()

This method returns the complete path of the current working directory for a script.

.flock(stream,mode)

This method locks or unlocks a file for the simultaneous use of multiple processes. `stream` is the name of a file to use for locking. `mode` may be LOCK_EX, LOCK_SH, LOCK_UN, or LOCK_NB.

.mkdir(dirpath)

This method creates the directory specified in the parameter `dirpath`. The specified directory may be an absolute or relative path specification.

.rmdir(dirpath)

This method removes the directory specified by the parameter `dirpath`.

Sorting

.bsearch(key, arrayToSort, [elementCount,] compareFunction)

This method looks for an array variable that matches `key`, returning it if found and `null` if not. It will only search through positive array members (i.e., array members with negative indices will be ignored). `compareFunction` must receive the key variable as its first argument and a variable from the array as its second argument. If `elementCount` is not supplied then will search the entire array. `elementCount` is limited to 64K for 16-bit version of ScriptEase.

The following example demonstrates the use of .`qsort`() and `bsearch`() to locate a name and related item in a list:

```
// create array of names and favorite food
list =
{
          { "Brent",   "salad" },
          { "Laura",   "cheese" },
          { "Alby",    "sugar" },
          { "Josh",    "anything" },
          { "Aiko",    "cats" },
          { "Quinn",   "anything from the garbage" }
};

// sort the list
Clib.qsort(list, ListCompareFunction);
Clib.Key[0] = "brent";
// search for the name Brent in the list
Found = bsearch(Key, list, ListCompareFunction);
// display name, or not found
if ( Found != null )
          Clib.printf("%s's favorite food is %s\n", Found[0][0],
                      Found[0][1])
else
          Clib.puts("Could not find name in list.");

function ListCompareFunction(Item1, Item2)
{
          return Clib.strcmpi(Item1[0], Item2[0]);
}
```

.qsort(array, [elementCount,] CompareFunction)

This method sorts elements in an `array`, starting from index 0 to `elementCount`-1. If `elementCount` is not supplied then will sort the entire array. This method differs from the `Array.sort`() method in that it can sort automatically-created arrays, whereas `Array.sort`() only works with arrays explicitly created with a `new Array` statement.

ElementCount is limited to 64K

The following code would print a list of colors sorted reverse-alphabetically and case-insensitive:

```
// initialize an array of colors
var colors = { "yellow", "Blue", "GREEN", "purple", "RED",
              "BLACK", "white", "orange" };
// sort the list using qsort and our ColorSorter routine
Clib.qsort(colors,"ReverseColorSorter");
// display the sorted colors
for ( var i = 0; i <= getArrayLength(colors); i++ )
          Clib.puts(colors[i]);

function ReverseColorSorter(color1,color2)
          // do a simple case insensitive string
          // comparison, and reverse the results too
{
          var CompareResult = Clib.stricmp(color1,color2)
          return( -CompareResult );
}
```

The output of the above code would be:

```
yellow
white
RED
purple
orange
GREEN
Blue
BLACK
```

Environment variables

.getenv([variableName])

If the parameter `variableName` is supplied, this method returns the value of a similarly named environment variable as a string if the variable exists, and null if VariableName does not exist. If no name is supplied then returns an array of all environment variable names, ending with a null element.

The following code would print the existing environment variables, in "EVAR=Value" format, sorted alphabetically.

```
// get array of all environment variable names
var EnvList = Clib.getenv();
// sort array alphabetically
Clib.qsort(EnvList, getArrayLength(EnvList)-1, stricmp);
// display each element in ENV=VALUE format
for ( var lIdx = 0; EnvList[lIdx]; lIdx++ )
          Clib.printf("%s=%s\n",EnvList[lIdx],
          Clib.getenv(EnvList[lIdx]));
```

.putenv(varName, stringValue)

This method sets the environment variable `varName` to the value of `stringValue`. If stringValue is `null` then `varName` is removed from the environment. For those operating

systems for which ScriptEase can alter the parent environment (DOS, or OS/2 when invoked with SESet.cmd or using .eSet()) the variable setting will still be valid when ScriptEase exits; otherwise the variable change applies only to the ScriptEase code and to child processes of the ScriptEase program. Returns -1 if there is an error, else 0.

Character classification

JavaScript does not have a true character type. For the character classification routines, a char is actually a one character string. Thus, actual programming usage is very much like C. For example, in the following fragment both `.isalnum()` statements work properly.

```
var t = Clib.isalnum('a');
Screen.writeln(t);

var s = 'a';
var t = Clib.isalnum(s);
Screen.writeln(t);
```

This fragment displays the following.

```
true
true
```

In the following fragment both `.isalnum()` statements cause errors since the arguments to them are strings with more than one character.

```
var t = Clib.isalnum('ab');
Screen.writeln(t);

var s = 'ab';
var t = Clib.isalnum(s);
Screen.writeln(t);
```

All character classification methods return booleans: `true` or `false`.

.isalnum(char)

Returns `true` if `char` is a character in one of the following sets: A-Z, a-z, or 0-9.

.isalpha(char)

Returns `true` if `char` is a alphabetic character in one of the following sets of characters: A-Z or a-z.

.isascii(char)

Returns `true` if `char` is an ASCII character in the following set of codes: 0-127.

.iscntrl(char)

Returns `true` if `char` is a control character in the set of ASCII characters. Control characters are in one of the following sets of codes: 0-31 or 127.

.isdigit(char)

Returns `true` if `char` is a decimal digit in the following set of characters: 0-9.

.isgraph(char)

Returns `true` if `char` is a printable character excluding the space character, code 32.

.islower(char)

Returns `true` if `char` is a lowercase character in the following set of characters: a-z.

.isprint(char)

Returns `true` if `char` is a printable character in the following set of codes: 32-126.

.ispunct(char)

Returns `true` if `char` is a punctuation character in one of the following sets of codes: 32-47, 58-63, 91-96, or 123-126.

.isspace(char)

Returns `true` if `char` is a white space character, that is, one of the following codes: 9, 10, 11, 12, 13, or 32 (horizontal tab, new line, vertical tab, form feed, carriage return, or space).

.isupper(char)

Returns `true` if `char` is an uppercase character in the following set of characters: A-Z.

.isxdigit(string)

Returns `true` if `char` is a hexadecimal digit in one of the following sets of characters: 0-9, A-Z, or a-z.

String manipulation

.rsprintf(formatString...)

This method returns a formatted string. It is exactly like .`printf`(), except that the string is returned instead of printed. For example, if in a script you had a line:

```
Clib.printf("%s has seen %s %d times.\n", name, movie,
            timesSeen);
```

and you wanted to pass the resulting string as a parameter to a function, you could do it like as follows.

```
func(Clib.rsprintf("%s has seen %s %d times.\n", name, movie,
                   timesSeen));
```

The following two lines of code achieve the same results, that is, create a string named `word` that contains the string "Who is #1?".

```
word = rsprintf("Who is #%d?", 3-2);
sprintf(word, "Who is #%d?", 3-2);
```

.rvsprintf(formatstring, valist)

This method returns formatted output using the variable argument list represented by the

parameter valist. This method is similar to .sprintf() except that it takes a variable argument list and returns a formatted string based on the arguments, rather than storing it in a string buffer. See .sprintf() and .va_start() for more information. The method .rvsprintf() returns a string specified by `format String` on success, else EOF on error.

.sscanf(formatString,...)

This method reads input from the standard input stream (the keyboard unless some other file has been redirected as `stdin`) and stores the data read in the variables provided as parameters following the `format String.` The data will be stored according to the character combinations in `format String` indicating how the input data is to be read and stored. This method is identical to calling .fscanf() with `stdin` as the first parameter. It returns the number of input items assigned; this number may be fewer than the number of parameters requested if there was a matching failure. If there was a conversion failure, EOF will be returned.

`format String` specifies the admissible input sequences, and how the input is to be converted to be assigned to the variable number of arguments passed to this function (in Windows, the input will not be read until it has been entered by hitting return).

Characters from input are matched against the characters of the `format String` until a percent character, `%`, is reached. `%` indicates that a value is to be read and stored to subsequent parameters following `format string`. Each subsequent parameter after `format String` gets the next parsed value taken from the next parameter in the list following format.

A parameter specification takes this form (angled brackets indicate required fields and square brackets indicate optional fields):

```
%[*][width]<type>
```

Where:

`*`	Suppress assigning this value to any parameter
`width`	maximum number of characters to read; fewer will be read if whitespace or nonconvertible character

type may be:

`d,D,i,I`	signed integer
`u,U`	unsigned integer
`o,O`	octal integer
`x,X`	hexadecimal integer
`f,e,E,g,G`	floating point number
`c`	character; if width was specified this will be an array of characters
`s`	string
`[abc]`	string consisting of all characters within brackets; A-Z

	represents range `"A" to "Z"`
`[^abc]`	string consisting of all character NOT within brackets

Returns EOF if input failure before any conversion occurs; otherwise it returns the number of variables assigned data.

.sprintf(string, formatString...)

This method writes output to the `string` variable specified by buffer according to `formatString`, and returns the number of characters written into buffer or EOF if there was an error. `formatString` can contain character combinations indicating how following parameters may be written. `string` need not be previously defined; it will be created large enough to hold the result. See `printf`() for a description of `formatString`.

The format string may contain character combinations indicating how following parameters are to be treated. Characters are printed as read to standard output until a percent character, `%`, is reached. `%` indicates that a value is to be printed from the parameters following the format string. Each subsequent parameter specification takes from the next parameter in the list following format (angled brackets represent required fields, while square brackets represent optional fields).

```
%[flags][width][.precision]<type>
```

flags may be:

`-`	Left justification in the field with blank padding; else right justifies with zero or blank padding
`+`	Force numbers to begin with a plus (+) or minus (-)
`blank`	Negative values begin with a minus (-); positive values begin with a `blank`
`#`	Convert using the following alternate form, depending on output data type:
`c,s,d,i,u`	No effect
`o`	0 (zero) is prepended to non-zero output
`x,X`	0x, or 0X, are prepended to output
`f,e,E`	Output includes decimal even if no digits follow decimal
`g,G`	same as e or E but trailing zeros are not removed

width may be:

`n`	(n is a number e.g., 14) At least n characters are output, padded with blanks
`0n`	At least `n` characters are output, padded on the left with zeros
`*`	The next value in the argument list is an integer specifying the output width

`.precision`	If precision is specified, then it must begin with a period (.), and may be as follows:
.0	For floating point type, no decimal point is output
.n	n characters or n decimal places (floating point) are output
.*	The next value in the argument list is an integer specifying the precision width.

type may be:

`d,i`	signed integer
`u`	unsigned integer
`o`	octal integer
`x`	hexadecimal integer with 0-9 and a, b, c, d, e, f
`X`	hexadecimal integer with 0-9 and A, B, C, D, E, F
`f`	floating point of the form [-]dddd.dddd
`e`	floating point of the form [-]d.ddde+dd or [-]d.ddde-dd
`E`	floating point of the form [-]d.dddE+dd or [-]d.dddE-dd
`g`	floating point of `f` or `e` type, depending on precision
`G`	floating point of `f` or `e` type, depending on precision
`c`	character ('a', 'b', '8', e.g.)
`s`	string

To include the `%` character as a character in the format string, you must use two `%` together (`%%`) to prevent the computer from trying to interpret it as one of the above.

Each of the following lines shows a sprintf example followed by the resulting string.

```
Clib.sprintf(testString, "I count: %d %d %d.",1,2,3)
     "I count: 1 2 3"
var a = 1;
var b = 2;
Clib.sprintf(testString, "%d %d %d",a, b, a+b)
     "1 2 3"
```

.strchr(string, char)

This method searches the parameter string for the character `char`. It returns a variable indicating the first occurrence of `char` in `string`, else it returns `null` if `char` is not found in string.

The following code fragment:

```
var str = "I can't stand soggy cereal."
var substr = Clib.strchr(str, 's');
Clib.printf("str = %s\n", str);
Screen.writeln("substr = " + substr);
```

results in the following output.

```
str = I can't stand soggy cereal.
```

```
substr = stand soggy cereal.
```

.strcmpi(string1, string2)

This method makes a case-insensitive comparison of the bytes of `string1` against `string2` until there is a mismatch or the terminating `null` byte is reached.

Returns result of comparison, which will be:

< 0	if `string1` is less than `string2`
= 0	if `string1` is the same as `string2`
> 0	if `string1` is greater than `string2`

.strcspn(string, charSet))

This method searches the parameter `string` for any of the characters in the string `charSet` , and returns the offset of that character. If no matching characters are found, it returns the length of the string. This method is similar to .`strpbrk`(), except that `strpbrk` returns the string beginning at the first character found, while `strcspn` returns the offset number for that character.

The following fragment demonstrates the difference between .strcspn() and .strpbrk().

```
var string="There's more than one way to skin a cat.";
var rStrpbrk = Clib.strpbrk(string, "dxb8w9k!");
var rStrcspn = Clib.strcspn(string, "dxb8w9k!");
Clib.printf("The string is: %s\n", string);
Clib.printf("\nstrpbrk returns a string: %s\n", rStrpbrk);
Clib.printf("\nstrcspn returns an integer: %d\n", rStrcspn);
```

And results in the following output:

```
The string is: There's more than one way to skin a cat.
strpbrk returns a string: way to skin a cat.
strcspn returns an integer: 22
```

.stricmp(string1, string2)

Same as `strcmpi.`

.strncat(&destString, sourceString, MaxLen)

This method appends up to MaxLen characters of `sourceString` string onto the end of `destString`. Characters following a `null`-byte in `sourceString` are not copied. The `destString` array is made big enough to hold `Clib.min(Clib.strlen(sourceString), MaxLen)`. The value of `destString` is returned.

.strncmp(string1, string2, MaxLen)

This method compares up to `MaxLen` bytes of `string1` against `string2` until there is a mismatch or reach the end of a string. The comparison ends when `MaxLen` bytes have been compared or when a terminating `null`-byte has been compared, whichever comes first. Returns result of comparison, which will be:

< 0	if `string1` is less than `string2`
= 0	if `string1` is the same as `string2`
> 0	if `string1` is greater than `string2`

.strncmpi(string1, string2, MaxLen)

This method compares up to `MaxLen` bytes of `string1` against `string2` until there is a mismatch or reach the end of a string. This method does a case-insensitive comparison, so that "A" and "a" are considered to be the same. The comparison ends when `MaxLen` bytes have been compared or when an end of string has been reached, whichever comes first. Returns result of comparison, which will be:

< 0	if `string1` is less than `string2`
= 0	if `string1` is the same as `string2`
> 0	if `string1` is greater than `string2`

.strncpy(&destString, sourceString, MaxLen)

This method copies `Clib.min(Clib.strlen(sourceString)+1, MaxLen)` characters from `sourceString` to `destString`. If `destString` is not already defined then this defines it as a string. `destString` is `null`-padded if `MaxLen` is greater than the length of `sourceString`, and a `null`-byte is appended to `destString` if `MaxLen` bytes are copied. It is safe to copy from one part of a string to another part of the same string. Returns the value of `dest`; that is, a variable into the `dest` array based at `dest[0]`.

.strnicmp(string1, string2, MaxLen)

The method .strnicmp() is the same as .strncmpi(). Please see .strncmpi() for more information

.strpbrk(string, charSet)

This method searches `string` for any of the characters in `charSet`, and returns the string based at the found character. Returns `null` if no character from charSet is found.

See .strcspn() for an example using this function.

.strrchr(string, char)

This method searches `string` for the last occurrence of `char`. The search is in the reverse direction, from the right, for `char` in `string`. The method returns a variable indicating the first occurrence of `char` in `string`, else it returns `null` if `char` is not found in `string`.

The following code:

```
var str = "I can't stand soggy cereal."
var substr = Clib.strrchr(str, 's');
Clib.printf("str = %s\n", str);
Screen.writeln("substr = " + substr);
```

Results in the following output.

```
str = I can't stand soggy cereal.
substr = soggy cereal.
```

.strspn(string, charSet)

This method searches `string` for any characters that are not in `charSet`, and returns the offset of the first instance of such a character. If all characters in `string` are also in `charSet` will return the length of `string`.

.strstr(string1, string2)

This method searches `string1`, starting at `string1[0]`, for the first occurrence of `string2`. The search is case-sensitive. The method returns a variable indicating the beginning of the first occurrence of `string2` in `string1`, else it returns `null` if `string2` is not found in `string1`.

The following script:

```
cFunction main()
{
          var Phrase = "To be or not to be? Beep beep!";
          do
              {
                  Screen.writeln(Phrase);
                  Phrase = Clib.strstr(Phrase + 1, "be");
              } while (Phrase != null);
}
```

results in the following output.

```
To be or not to be? Beep beep!
be or not to be? Beep beep!
be? Beep beep!
beep!
```

.strstri(string, substring)

This method searches `string`, starting at `string[0]`, for the first occurrence of `substring`. Comparison is case-sensitive. This is a case-insensitive version of the `String.substring()` method. Returns `null` if `substring` is not found anywhere in `string`; otherwise returns variable for the `string` array based at the first offset matching `substring`.

.strtok(&sourceString, delimiterString)

`sourceString` is a string that consists of text tokens (substrings) separated by delimiters found in `delimiterString`. Bytes of `sourceString` may be altered during the first

and subsequent calls to `strtok()`.

On the first call to `strtok()`, `sourceString` points to the string to tokenize and `delimiterString` is a list of characters which are used to separate tokens in source. This first call returns a variable pointing to the `sourceString` array and based at the first character of the first token in `sourceString`. On subsequent calls, the first argument is `null` and `strtok` will continue through `sourceString` returning subsequent tokens.

The initial var must remain valid throughout following calls with `null` as the initial var. If you change the string in any way, a following call to `strtok()` must be of the first syntax form (i.e., the new string must be passed as a first parameter). Returns `null` if there are no more tokens; otherwise returns sourceString array variable based at the next token in sourceString.

The following code:

```
var source = " Little   John,,,Eats ?? crackers;;;! ";
var token = Clib.strtok(source,", ");
while( null != token )
{
          Clib.puts(token);
          token = Clib.strtok(null,";?, ");
}
```

produces the following list of tokens.

```
Little
John
Eats
crackers
!
```

.toascii(char)

This method translates the parameter `char` to ASCII format, i.e., to seven bits.

The translation is done by clearing all but the lowest 7 bits. The return is `char` converted to ASCII.

.vsprintf(&buffer, formatstring, valist)

This method puts formatted output into `buffer`, a string, using a variable number of arguments, specified by `valist`. The parameter `formatstring` specifies that format of the data put into `buffer`. This method is similar to `.sprintf()` except that it takes a variable argument list. See `sprintf()` and `.va_start()` for more information.

The method returns the number of characters written to buffer on success, else EOF on error.

Memory manipulation

.memchr(buf, char[, size])

This method searches buf, a buffer, and returns a variable indicating the first occurrence of byte `char`. If the parameter `size` is not specified, then the method searches the entire buf from element zero. The return is `null` if char is not found in array, else the return is a buffer which has its offset 0 nt the first `char` in `buf`.

.memcmp(buf1, buf2[, len])

This method compares the first len bytes of `buf1` and `buf2`. If the parameter `len` is not specified, then len is the smaller of the lengths of `buf1` and `buf2`. If `len` is specified and one of the buffers is shorter than the specified length, then ScriptEase treats length of the shorter buffer as being `len`.

The method returns the result of a comparison. The return conforms to one of the following.

< 0 if `buf1` is less than `buf2`

= 0 if `buf1` is the same as buf2 for len bytes

> 0 if `buf1` is greater than buf2

.memcpy(&DestBuf, SrcBuf[, len])

This method copies the number of bytes specified by len from `ScrBuf` to `DestBuf`. If `DestBuf` is not already defined, then it is defined as a buffer. If the parameter `len` is not supplied, then all of the bytes in `SrcBuf` are copied to `DestBuf`.

ScriptEase ensures protection from data overwrite, so in ScriptEase the `.memcpy()` method is the same as `.memmove()`.

.memmove(DestBuf, SrcBuf[, len])

ScriptEase ensures protection from data overwrite, so in ScriptEase the .memmove() method is the same as .memcpy().

.memset(buf, char[, len])

This method sets the first number, as specified by `len`, of bytes of `buf` to character `char`. If `buf` is not already defined, then it is defined as a byte buffer of size `len`. If the length of `buf` is less than the number of bytes specified by `len`, then `buf` is grown to be big enough for `len` bytes. If the parameter `len` is not supplied, then len is the size of `buf`, starting at index 0.

Math

.cosh(x)

This method returns the hyperbolic cosine of x.

.div(numerator, denominator)

This method performs integer division and returns a quotient and remainder in a structure. Since integers and long integers are the same in ScriptEase, .div() is the same as .ldiv(). The value returned is a structure with the following elements, which are the result of dividing numerator by denominator.

`.quot`	quotient
`.rem`	remainder

.fabs(x)

This method returns the absolute, non-negative, value of a float x.

.frexp(x, exponent)

This method breaks x into a normalized mantissa between 0.5 and 1.0 and calculates an integer exponent of 2 such that `x == mantissa * 2 ^ exponent`. The return is normalized mantissa between 0.5 and 1.0, or 0.

.labs(x)

This method returns the absolute, non-negative, value of an integer.

Since integers and long integers are the same in ScriptEase, .labs() is the same as .abs().

.ldexp(mantissa, exponent)

This method is the inverse of .frexp() and calculates a floating point number from the following equation:

`mantissa * 2 ^ exponent.`

The return is the result of the previous calculation

.ldiv(numerator, denominator)

This method performs integer division and returns a quotient and remainder in a structure. Since integers and long integers are the same in ScriptEase, .ldiv() is the same as .div(). The value returned is a structure with the following elements, which are the result of dividing numerator by denominator.

`.quot`	quotient
`.rem`	remainder

.modf(x, i)

This method splits a floating point number `x` into integer and fractional parts, where the integer and fraction both have the same sign as `x`. The method sets the parameter `i` to the integer part of `x` and returns the fractional part of `x`.

.rand()

This method generates a random number between 0 and RAND_MAX, inclusive. The sequence of pseudo-random numbers is affected by the initial generator seed and by earlier calls to `.rand()`. See `.srand()` for information about the initial generator seed. This method returns pseudo-random number between 0 and RAND_MAX, inclusive.

.sinh(x)

This method returns the hyperbolic sine of the float x

.srand[seed]

This method initializes a random number generator using the parameter `seed`. If seed is not supplied, then a random `seed` is generated in an a manner that is specific to different operating systems.

.tanh(x)

This method calculates and returns the hyperbolic tangent of the parameter `x`, a float.

Variable argument lists

.va_arg([valist] (,) [offset])

The method .va_arg() provides an alternate way to retrieve a function's parameters. It's most often used when the number of parameters passed to the function is not constant. This method covers the same territory as the Function.arguments[] property and is provided for those used to C's functions for handling variable arguments.

When called with no parameters, `.va_arg` returns the number of parameters passed to the current function. If an offset is supplied, `.va_arg` returns the input variable at index: offset. `.va_arg(0)` is the first parameter passed, `.va_arg(1)` the second, etc. It is a fatal error to retrieve an argument offset beyond the number of parameters in the function or the `valist`.

The valist form (with an optional offset) uses a valist variable that has been previously initialized with `va_start()`. Each call to `va_arg(valist)` returns the next parameter passed to the function. If an offset is passed in the variable at that offset from the original starting place of the valist will be returned.

The following script:

```
function main()
{
          lips(0, 1, 2, 3, 4)
}

lips()
{
          Clib.va_start(valist)
          Clib.printf("va_arg(0) = %d\n", Clib.va_arg(0));
          Clib.printf("va_arg(1) = %d\n", Clib.va_arg(1));
          Clib.printf("va_arg(valist) = %d\n",
              Clib.va_arg(valist));
          Clib.printf("va_arg(valist, 2) = %d\n",
              Clib.va_arg(valist, 2));
          Clib.printf("va_arg(valist, 2) = %d\n",
              Clib.va_arg(valist, 2));
          Clib.printf("va_arg(valist) = %d\n",
              Clib.va_arg(valist));
          Clib.getch()
}
```

produces the following output:

```
va_arg(0) = 0
va_arg(1) = 1
va_arg(valist) = 0
va_arg(valist, 2) = 3
va_arg(valist, 2) = 3
va_arg(valist) = 1
```

.va_start(va_list[, InputVar])

This method initializes `va_list` for a function with variable or unknown number of arguments. After this call, may be used in further calls to `va_arg()` to get the next argument(s) passed to the function.

The parameter `InputVar` must be one of the parameters defined on the function line; the first argument returned by the first call to `va_arg()` will be the variable passed after `InputVar`. If `InputVar` is not provided, then the first parameter passed to the function will be the first one returned by `va_arg(va_list)`.

Returns the number of valid calls to `va_arg(va_list)`, i.e., how many variables are available in this `va_list`.

The following example uses and accepts a variable number of strings and concatenates them all together.

```
function MultiStrcat(Result,InitialString);
// Append any number of strings to InitialString.
// e.g., MultiStrcat(Result,"C:\\","FOO",".","CMD")
{
          Clib.strcpy(Result,""); // initialize the result;
          var Count = Clib.va_start( ArgList, InitialString );
          for ( var i = 0; i < Count; i++ )
              Clib.strcat(Result, va_arg(ArgList) );
}
```

.vfprintf(filePointer, formatString, va_list)

This method formats a string with a variable number of arguments and prints it to the file specified by `filePointer`. It returns the number of characters written, or a negative number if there was an output error. See `.fprintf()` and `.sprintf()` for more details.

.vfscanf(filePointer, formatString, va_list)

This method is similar to `.fscanf()` except that it takes a variable argument list (see `.va_start()`). See `.fscanf()` for more details.

.vsscanf(buffer, format, valist)

This method is similar to `.sscanf()` except that it takes a variable argument list (see `.va_start()`). The parameters following the format string will be assigned values according to the specifications of the format string.

The function returns the number of input items assigned. This number may be fewer than the number of parameters requested if there was a matching failure. See .sscanf() for more details.

Redundant functions in the Clib object

The Clib Object includes the functions from the C standard library. As a result, some of the methods in the Clib Object overlap methods in JavaScript. In most cases, the newer JavaScript methods should be preferred over the older C functions. However, there are times, such as when working with many cfunctions or with string routines that expect `null` terminated strings, that the Clib methods make more sense and are more consistent in a section of a script.

The Clib methods list below are paired with equivalent methods in JavaScript. Since ScriptEase, JavaScript and the ECMAScript standard are developing and growing, generally, a programmer should favor the JavaScript methods over equivalent methods in the Clib object.

Clib method	Description	JavaScript method
.abs()	Calculate absolute value	Math.abs()
.acos()	Calculate the arc cosine.	Math.acos()
.asin()	Calculate the arc sine.	Math.asin()
.atan()	Calculate the arc tangent.	Math.atan()
.atan2()	Calculate the arc tangent of a fraction	Math.atan2()
.atof()	Convert string to float.	Automatic conversion
.atoi()	Convert string to integer.	Automatic conversion
.atol()	Convert string to long.	Automatic conversion
.ceil()	Round up number to nearest integer.	Math.ceil()
.cos()	Calculate the cosine	Math.cos()
.exp()	Compute the exponential function.	Math.exp()
.floor()	Round number down to nearest integer.	Math.floor()
.fmod()	Calculate remainder.	% operator, modulo
.log()	Calculate natural logarithm	Math.log()
.max()	Return the largest of one or more values.	Math.max()
.min()	Return the smallest of one or more values.	Math.min()
.pow()	Calculates x to the power of y	Math.pow()
.sin()	Calculate the sine.	Math.sin()
.sqrt()	Calculate the square root	Math.sqrt()

Clib method	Description	JavaScript method
.strcat()	Append one string to another	+ operator
.strcmp()	Compare two strings	== operator
.strcpy()	Copy a string	= operator
.strlen()	Get length of string	String.length
.strlwr()	Convert string to lower case.	String.toLowerCase
.strtod()	Convert string to decimal	Automatic conversion
.strtol()	Convert string to long	Automatic conversion
.strupr()	Convert string to upper case.	String.toUpperCase
.tan()	Calculate the tangent	Math.tan()
.tolower()	Convert character to lower case.	String.toLowerCase
.toupper()	Convert character to upper case.	String.toUpperCase
.va_end()	End of va_list. This method does nothing and may be omitted.	

The Blob Object

This section describes BLObs, Binary Large Objects.

Blob.get()

Syntax

```
byte          Blob.get(BLOb BlobVar, int offset, int DataType)
int           Blob.get(BLOb BlobVar, int offset, int DataType)
float         Blob.get(BLOb BlobVar, int offset, int DataType)
byte[]        Blob.get(BLOb BlobVar, int offset, int bufferLen)
object        Blob.get(BLOb BlobVar, int offset,BlobDescriptor,
              DataDefinition)
```

Description

This method reads data from a specified location of a Binary Large Object, a BLOb and is the companion function to `Blob.put()`. The parameter `BlobVar` specifies the BLOb to use. The parameter offset specifies where in the BLOb to get data. The last parameter specifies the format of the data in the BLOb and, hence, determines the type of the value returned which is the data read from the BLOb.

Valid values for DataType are:

UWORD8, SWORD8, UWORD16, SWORD16, UWORD24, SWORD24, UWORD32, SWORD32, FLOAT32, FLOAT64, or FLOAT80

See `Clib.fread()` for more information on these DataType values.

Blob.put()

Syntax

```
int    Blob.put(BLOb BlobVar[, int offset], Var v, int
                  DataType)
int    Blob.put(BLOb BlobVar[, int offset], byte[] buffer,
                  bufferLen)
int    Blob.put(BLOb BlobVar[, int offset], object SrcStruct,
                  blobdescriptor DataDefinition)
```

Description

This method puts data into a specified location of a Binary Large Object, BLOb and, along with `Blob.get()`, allows for direct access to memory within a var. The contents of such a variable may be viewed as a packed structure. Data can be placed at any location within a BLOb. The parameter `BlobVar` specifies the BLOb to use. The parameter `offset` specifies where, in the BLOb, to write data. The third parameter is the data to write. The last parameter specifies the format of the data in the BLOb.

`Blob.put()` returns the byte offset for the next byte following the section where data was just put. If the data is put at the end of the BLOb, then the return is equivalent to the size of the BLOb.

If `offset` is not supplied, then the data is put at the end of the BLOb, or at offset 0 if the BLOb is not yet defined.

The data in `v` is converted to the specified `DataType` and then copied into the bytes specified by offset.

If `DataType` is not the length of a byte buffer, then it must be one of these types:

UWORD8, SWORD8, UWORD16, SWORD16, UWORD24, SWORD24, UWORD32, SWORD32, FLOAT32, FLOAT64, or FLOAT80

See `Clib.fread()` for more information on these DataType values.

Example

If you were sending a pointer to data in an external C library and knew that the library expected the data in a packed C structure of the form:

```
struct foo
{
          signed char a;
          unsigned int b;
          double c;
};
```

and if you were building this structure from three corresponding variables, then such a building function might look like the following:

```
function BuildFooBlob(a, b, c)
{
          var offset = Blob.put(foo, 0, a, SWORD8);
          offset = Blob.put(foo, offset, b, UWORD16);
          Blob.put(foo, offset, c, FLOAT64);
          return foo;
}
```

or, if an offset were not supplied:

```
functionBuildFooBlob(a, b, c)
{
          Blob.put(foo, a, SWORD8);
          Blob.put(foo, b, UWORD16);
          Blob.put(foo, c, FLOAT64);
          return foo;
}
```

Blob.size()

Syntax

```
int Blob.size(BLOb BlobVar[, SetSize])
int Blob.size(int DataType)
int Blob.size(int bufferLen)
int Blob.size(blobDescriptor Definition)
```

Description

This method determines the size of a Binary Large Object, BLOb. The parameter `BlobVar` specifies the BLOb to use. If `SetSize` is provided, then the BLOb `BlobVar` is altered to this size or created with this size.

If DataType, `bufferLen`, or `Definition` are used, `Blob.size()` returns the size of a BLOb that would contain the type of data item used in by `Blob.get()` or `Blob.put()`. In these cases, these parameters specify the type to be used for converting ScriptEase data to and from a BLOb.

`Blob.size` returns the size of a BLOb which is the number of bytes in `BlobVar`. If `SetSize` is supplied, then the return is `SetSize`.

The blobDescriptor Object

When an Object needs to be sent to a process other than the ScriptEase interpreter, such as to a Windows API function, a blobDescriptor Object must be created that describes the order and type of data in the Object. This description tells how the properties of the Object are stored in memory and is used with functions like Clib.fread() and SElib.dynamicLink().

A blobDescriptor has the same data properties as the Object it describes. Each property must be assigned a value that specifies how much memory is required for the data held by that property. Consider the following Object.

```
Rectangle(width, height)
{
          this.width = width;
          this.height = height;
}
```

The following code creates a blobDescriptor object that describes the Rectangle Object:

```
var bd = new blobDescriptor();

bd.width  = UWORD32;
bd.height = UWORD32;
```

You can now pass `bd` as a blobDescriptor parameter to functions that require one. The values assigned to properties will depend on what the receiving function expects. In the example above, the function called expects to receive an Object that contains two 32-bit words or data values. If you write a blobDescriptor for a function that expects to receive an Object containing two 16-bit words, assign the two properties a value of UWORD16.

The following values may be used for blobDescriptors.

`UWORD8`	Stored as a byte
`SWORD8`	Stored as an integer
`UWORD16`	Stored as an integer
`SWORD16`	Stored as an integer
`UWORD24`	Stored as an integer
`SWORD24`	Stored as an integer
`UWORD32`	Stored as an integer
`SWORD32`	Stored as an integer
`FLOAT32`	Stored as a float
`FLOAT64`	Stored as a float
`FLOAT80`	Stored as a float (not available in Win32)
`STRINGHOLDER`	Used to indicate that a value that is assigned a string by the function to which it is passed. It allocates 10,000 bytes to contain the string, and then truncates this length to the appropriate size, removes any terminating `null` characters, and initializes the properties of the string.

If the blobDescriptor describes an Object property that is a string, the corresponding property should be assigned a numeric value that is larger than the length of the longest string the property may hold. Object methods usually may be omitted from a blobDescriptor.

DOS / WIN16

.address(segment, offset)

DOS, 16-bit Windows

This method converts the segment and offset into a single segment:offset address. It returns a segment:offset address suitable for use in calls such as .peek() and .poke().

.asm(byte[] buf[, ax[, bx[, cx[, dx[, si[, di[, ds[, es]]]]]]]])

DOS, 16-bit Windows, OS/2

This function sets the registers and executes the code contained in `buf`, which must be a buffer containing assembly language code. The optional variables `ax`, `bx`, etc. are the values to be put into the registers before executing the code; in 16-bit systems they will be 16-bit values, and 32-bit values in 32 bit systems.

asm() makes a far call to whatever routine that you have coded into `buf`. `ax`, `bx`, `cx`, `dx`, `si`, `di`, `ds`, and `es` are optional; if some or all are supplied, then the `ax`, `bx`, `cx`, etc... will be set to these values when the code at `buf` is called. The code in `buf` will be executed with a far call to that address. The ScriptEase calling code will restore ALL registers except ss, sp, ax, bx, cx, and dx. If `es` or `ds` are supplied, then they must be valid values or 0 to use the current value. This function returns a long value for whatever is in DX:AX when buf returns.

Warning: Please read the note concerning this function at the beginning of this section.

.inport(portID)

This method reads a byte from the hardware port indicated by portID.

.inportw(portID)

This method reads a word (16 bit) from the hardware port indicated by portID. The value read is unsigned (not negative).

.outport(int portID, byte value)

This method writes a byte value to the hardware port indicated by portID.

.outportw(int portID, int value)

This method writes a word (16-bit) value to the hardware port indicated by portID.

.interrupt(Interrupt, RegIn [, RegOut])

DOS, 16-bit Windows

Set registers, call 80x86 interrupt function, and then get the return values of the registers. `RegIn` and `RegOut` are objects containing properties corresponding to the registers on an 80x86. On input, the properties that are defined will be set, and those that are not defined will be set to zero, with the exception of the segment registers (ES & DS) which retain their current values if not explicitly specified. The possible defined input values are AX, AH, AL, BX, BH, BL, CX, CH, CL, DX, DH, DL, BP, SI, DI, DS, ES. All Fields of the Output reg structure are the same, with the addition of the FLAGS member, and all are set before returning. If RegOut is not supplied, then the return registers and FLAGS register will be set for RegIn on return from the interrupt call.

Since many interrupts set the carry flag for error, this function will return `False` if the carry flag is set, else returns `true`.

.offset(buffer)

DOS, 16-bit Windows

This method, along with its companion method `.segment()`, breaks a buffer or far pointer into its segment and offset components.

.segment(buf)

DOS, 16-bit Windows

This method and its companion method `.offset()` return the segment and offset of the data at index 0 of buf, which must be a buffer. The buffer must be already big enough for whatever purpose it is used, and no changes may be made to the size of the buffer after these values are determined because changing the size of the buffer might change its absolute address.

If the address versions are used, then address is assumed to be a far pointer to data, and segment will be the high word while address will be the low word. See `. address()` for converting segment and offset into a single address.

Return segment or offset of buffer such that 8086 would recognize the address segment::buffer as pointing to the first byte of `buf`.

OS/2

.eSet(fileSpec)

This method writes new environment variable settings to a file, returning `true` if it is successful and `false` if it is unable to write the settings to the file.

The parameter `fileSpec` is the name of the file to create, if necessary, and write to. When a call is made to `.putenv()`, as many statements of the form "SET VAR=Value" as necessary are written to the file so that ScriptEase always has an updated version of the variables. Note that this operation is unnecessary if the SE_ESET environment variable is set. In this case, the call to .eSet(%SE_ESET%) is automatically generated as the last statement before a smooth exit from ScriptEase.

.inport(portID)

This method reads a byte from the hardware port indicated by portID.

.inportw(portID)

This method reads a word (16 bit) from the hardware port indicated by portID. The value read is unsigned (not negative).

.outport(int portID, byte value)

This method writes a byte value to the hardware port indicated by portID.

.outportw(int portID, int value)

This method writes a word (16-bit) value to the hardware port indicated by portID.

.pmDynamicLink()

This method is identical to `.dynamicLink()` except that `.pmDynamicLink` passes DLL calls through the Seos2pm.exe gateway program. Seos2.exe is not a PM program but Seos2pm.exe is, so Seos2pm.exe can make these calls for Seos2.exe. Addresses and buffers are automatically transferred and shared between Seos2.exe and Seos2pm.exe, so in most cases your code does not need to concern itself with memory protection. The exception to this is if one of the arguments to `.pmDynamicLink()` contains or will receive a pointer. In this case you need to put or get that data explicitly by using `.pmPeek()` and `.pmPoke()` instead of the usual `.peek()` and `.poke()` routines.

.pmInfo()

See the SElib method `.info()`. This method, `.pmInfo()`, works identically but retrieves the information for the Seos2pm.exe gateway function. the method `.pmDynamicLink()` often must use these values instead of those from `.info()`.

.pmPeek()

This method is identical to `SElib.peek()`, but accesses memory that is given to Seos2pm.exe or that may only be accessible from a PM program.

.pmPoke()

This method is identical to `SElib.poke()` but accesses memory that is given to Seos2pm.exe or that may only be accessible from a PM program.

.processList([boolean IncludeThreadInfo])

This method returns an array of objects containing data for every running process in the system. If `IncludeThreadInfo` is `true`, then information is also added for each thread in each process. If `IncludeThreadInfo` is not supplied then `false` is assumed, so only process data is returned.

If there is an error in the thread information, this method returns `null`, otherwise it returns an array of objects containing the following properties.

`.id`	ProcessID for this process
`.parent`	ProcessID for the parent of this process
`.name`	String containing the full name of this process
`.threads`	Array of structures describing each thread of this process; this structure element will not be returned unless IncludeThreadInfo is set to equal *True*, in which case .threads is an array of structures for each thread in the process, where the thread structure contains these structure elements:
`.ProcID`	ID of this thread within the process
`.SysID`	ID of this thread within the system
`.Priority`	Running priority of this thread
`.Status`	Current running state of this thread

ScriptEase
Distributed Scripting Protocol

The Distributed Scripting Protocol, or DSP, is Nombas' method for remote scripting. Using DSP, your application can communicate with and run scripts on any other application that uses ScriptEase and is DSP-enabled. For instance, your application can be controlled by a ScriptEase: Desktop script running on another machine, a ScriptEase: Web Server Edition script, or even another instance of your application running on another machine.

There is quite a bit of information about distributed scripting we need to cover, and much of it is interrelated. For that reason, at times, related information to a topic isn't explored until a later section of the manual. It may take you a couple of reads to get the most from this manual.

I: Adding DSP to Your Application

The first part of this document is concerned with how you get DSP running on your application. The second part of this document is using DSP to script remote applications.

For your application to perform remote scripting tasks, you will first need to add the ScriptEase: ISDK to your application. This process is extensively detailed in the ScriptEase: ISDK manual. You can also use DSP with ScriptEase products such as ScriptEase: Desktop or ScriptEase: Web Server Edition. In any case, the external library capability of ScriptEase allows you to use (via the '#link' directive) the DSP external library. If you have decided to turn off the external library capability in your application, you will need to add the DSP library internally to your application. This is a simple process.

First, add the source file 'srclib/dsp/sedsp.c' to your builds. Second, edit your 'jseopt.h' file and add the following line to it:

```
#define JSE_DSP_ALL
```

That's it. Recompile and relink your application and you can now use the Distributed Scripting Protocol with it.

A common usage of DSP involves using the internet (TCP-IP) for tranporting the data, as detailed below. This is usually done in ScriptEase via the sesock external library. Again,

if you are not using external libraries, but still want to use sockets for your DSP transport, you need to add the socket library internally. This is done just like the sedsp functionality was added, by including the file 'srclib/socket/sesocket.c' and adding a define to your 'jseopt.h' file:

```
#define JSE_SOCKET_ALL
```

The Distributed Scripting Transport Mechanism

Distributed scripting consists of two parts, the protocol itself and the transport layer. The protocol is concerned with what information to send while the transport layer sees to it that the information gets passed between the two machines. The protocol is DSP itself; we've done all the work on implementing it, so you need only use it, which we'll discuss extensively later. The protocol is attached to no particular transport mechanism. You choose the one you want. This section is concerned with the transport layer. DSP needs to get particular information to the partner application. You need to get the applications connected and able to pass this information back and forth.

Commonly, you will use TCP-IP to allow applications to script each other over the internet. However, we have an example of using shared files to implement transport as well, and you can use any number of other ways to move the data between two machines. We include an implementation of TCP-IP tranport as well as shared files for you to use. Before we get into how you would write your own mechanism, we will explain how to use these two implementations

We will make all examples in this document scripted examples; you would use these when all DSP interaction is done inside scripts. For instance, in a client/server model, both the client and servers would be scripts running on a ScriptEase ISDK application. You can access the DSP routines directly from the ScriptEase API. We reserve discussion of this until Appendix i so as to not make the explanations we are giving confusing.

Shared Files

The shared file mechanism is implemented in the file 'dspfile.jsh', simply include this file at the top of your DSP script. Now you can create a new DSP connection, using shared files, using the 'new fileDSP()' constructor. This constructor takes two arguments, the name of two files used to pass the data back and forth. The result of this call is a full DSP connection object. Once you have this object, you can use all of the DSP mechanisms described later. You have your ticket, now you are ready see the show.

Here is an example of creating a DSP object using our shared file transport mechanism:

```
var conn = new fileDSP("dspfile.in","dspfile.out");
```

You will need to specify the same two files for both DSP partner applications. The only little gotcha is that the file names for the two partners must be swapped; the input file for

one application is the output file for the other application, and vice versa. In the example above, one application has set up file DSP. The other application will need to initialize its connection like this:

```
var conn = new fileDSP("dspfile.out","dspfile.in");
```

Obviously, shared files can only work on machines that can see the same files. This is easily done if both scripts are run on the same machine, or machines that share files such as via a Netware file server.

TCP-IP

With the popularity of the internet, it is only a natural that people want to use it to do distributed scripting. Many popular internet staples, such as web servers and email, are implemented at their core by a daemon that accepts connections on a TCP-IP socket and then allows a remote application to control it. With DSP, you can do exactly the same thing, but you have the full power of Javascript to allow far more complex commands than the simplistic ones most daemons allow. Once your application has integrated the ScriptEase ISDK, not only can you run scripts locally to control it, you can allow other machines to run them to control your application as well.

TCP-IP is the 'language' of the internet. ScriptEase provides the library calls necessary to create and use TCP-IP sockets in the 'sesock' external library. We provide an include file, "isdp.h", that uses these calls to implement a transport mechanism for DSP.

Like the file transport mechanism above, both partners must create their end of the connection. In the TCP-IP world of sockets, one partner must create a connection and the other partner must connect to that existing connection. This easily lends itself to a client-server model, but as we will see later, that is not the only possible model; once connected, either side may run scripts on the other side, or both may.

TCP-IP Server

We will first examine the listener side. This is the application that creates a socket to accept connections on and waits for those connections to occur. This application creates a socket it can listen on by using the 'new iDSPServer()' constructor. This is traditionally called a "server", but keep in mind that the client-server relationship we are discussing now is only for getting the two applications connected. Once the connection is established, either side may script the other as we will describe in Chapter II.

Here is the code snippet for creating a server:

```
#include "idsp"

   var server = new iDSPServer( port );
```

In this case, 'port' should be a numeric value, the port to listen on. When using sockets, you must specify a port number so that the connecting client can then find you. Other applications on your machine may be using their own ports to do TCP-IP connections, so it is slightly possible you may get a conflict. If this happens, select a different port number. You should always use port numbers that are 1000 or higher; the lower port numbers are reserved for standard system programs like mail and web servers.

Once you have done this, you will have a socket that you can accept connections on. This value returned is not itself a connection, it is just a socket handle with some extra DSP information attached. It is a permanent place for clients to find you and for to find the clients. You use this socket to get connections to clients; You use that handle's method 'accept()'. Even after you have a connection and are performing DSP scripting using it, the original socket remains. You can continue to accept more connections now and in the future using it. Usually, a server handling a TCP-IP socket runs indefinitely, until it is explicitly shutdown, and this socket exists for the life of the program.

Once we have created a socket to listen on using the above code, we need to accept connections to actually do DSP scripting. Whenever a TCP-IP client connects to us, we use the 'accept()' method of that socket to get this connection and service the client. We can use the 'ready()' method of the socket to know if such a connection has been made. If we just call 'accept()', it will always get a connection for us. If no one is trying to connect when we make a call to 'accept()', we will wait until someone does. This can be fine for some programs, but in others you may want your program to do other tasks rather than be put to sleep waiting for a connection to appear. This is especially important for Windows scripts. While the 'accept()' method is waiting, Windows messages are not processed, and the application will appear to freeze.

So, here is a simple fragment to get new connections using iDSP (this fragment builds on the last simple fragment):

```
/* This server just runs forever */
   while( 1 )
   {
      /* wait for the next connection to appear */
      while( !server.ready(250) );
      /* get the next connection */
      var conn = server.accept();

      /* do some DSP scripting with the remote host
*/

      /* close the connection and loop to service the
next connection */
      conn.dspClose();
   }
```

Notice that this simple server code handles only a single connection at a time. This is not necessary, you can call the 'accept()' method multiple times and have more than one connection simultaneously, if multiple clients are trying to connect to your server at the same time. Of course, the code to handle their requests simultaneously will be more complex. Handling multiple socket connections at once is demonstrated in the sample 'inndsp.jse', which is discussed fully in Appendix ii.

Of course, any of the methods you would use to handle multiple socket connections applies equally to multiple DSP connections, since at its heart, an iDSP connection is really just a socket. While a few of the socket library calls, such as 'accept()' and 'listen()' are directly mimicked by the iDSPServer() object, not all are. Fortunately, you can get the socket object itself by using the '.dspConnection' field of a DSP connection, such as 'server.connection' or 'conn.connection', in the sample code fragments above. You can then use this socket in any socket calls.

This sample code fragment uses the 'dspClose()' call. This will be discussed as part of the next chapter on how you actually do DSP scripting. Suffice it to say for now, when you finish scripting on a particular connection, you use 'dspClose()' to shut down the connection.

TCP-IP Client

Now that we have the server half of the applications, we need a client. The client will connect to an already existing server's socket. Fortunately, it is much simpler than the work we needed to do to get the server running. We can connect to an existing server like this:

```
var client = new iDSP(host,port);
```

The 'host' parameter is the internet machine name that the server is running on. For instance, my machine is named "outworld.nombas.com", so if you wanted to connect to a server running on my machine, that would be the name you use. You can use "localhost" to have the client connect to an application running on the same machine the client is on. 'port' is the port number, the same one that the server used when it initialized its socket. That was described above in the section on the server.

That's it. Again, both partner applications will have a connection they can use to script the other application. We will now dive into actually using those connections. If you would like to build a custom transport mechanism instead of using one of our sample ones, Appendix ii has full details on how you go about doing this.

II: Using Distributed Scripting

Distributed Scripting Models

After you have a connection established, remote scripting can take place. The connection does not have any preferred 'direction'. Either side can call DSP commands that the other will process.

For instance, the TCP-IP server may accept connections from clients, then run scripts on them. Maybe you are writing a server that contains updates for software installed on your company's machines. Each day, as part of a CRON job, every machine in the office connects to that server. The server then queries the client machine to see what updates it has that are not yet on the client machine, and installs them.

On the other hand, you can instead make the client machine run a script which queries the server to see what updates it has, then installs them on itself. Which partner is running the script and which is being scripted is determined by the scripts being run themselves. DSP allows both sides of the connection to run scripts on the other.

Let's look at two common models for doing DSP scripting.

Client-Server

Often, you'll want a Client-Server model. In this case, after the connection is made, one side will run a script that may execute script code on the other side. The other partner will just wait while that script runs. In this case, the side running the script is the client, and the side waiting is the server. The service being provided is the processing of the DSP scripts that the client wishes to run on the server. This is the simplest model and is very easy to understand. Again, don't confuse this with the iDSP client-server connections. Once the connection is established, either side may be the client as we are talking about now. It is most common that the side doing the connecting is the client, though.

There is nothing to say about the client in this section; it just runs some script doing DSP. What you can do with DSP will be fully described below. In this model, the client script is the only one 'doing anything'.

The server is also running a script. However, in this model, its only job is to wait for the client to finish while processing any DSP commands the client wants done. It is possible for DSP errors to occur, such as losing the connection, so the server's script needs to be able to handle that.

Here is a simple fragment that implements a DSP server:

```
      try
      {
         /* while the other guy has stuff to do, do
it */
         while( conn.dspService() );
      }
      catch( e )
      {
         /* if some error occurs, don't crash, just
terminate the connection */
      }
      conn.dspClose();
```

You would use this fragment after the connection is made, for instance after the line in the TCP-IP sample fragment that reads:

```
/* do some DSP scripting with the remote host */
```

The key routine we learn about here is 'dspService().' DSP commands sent from the remote host are processed with this call. Each call to it processes a single command. It returns 'false' if the remote host closed the connection using 'dspClose()'. We don't close our own connection until after the client has done all of the scripting it desires.

Peer-To-Peer Scripting

Not all DSP applications will want only one side running code. You may want both sides running code and using distributed scripting to communicate. In this case, we actually have a relationship among peers. This is the basic DSP model; the Client-Server really just is a special case of this in which one side, the server, does nothing. As peers, each side is free to run whatever script it wants. It can call functions and modify variables on the other side using DSP.

The main point to remember while doing peer-to-peer communication is that DSP requests from the other side are only handled when your own script calls some DSP function. If you do other scripting tasks, the other side will be waiting on any DSP processing it needs done until you actually call a DSP function. For this reason, peer-to-peer scripting is necessarily more complex than a Client-Server relationship. Several hints are in order. First, make sure your script periodically calls 'dspService()' to process any commands that your partner needs done. Second, you must have some mechanism to signal both sides that you are ready to quit. If one side calls 'dspClose()', then the connection will be lost, even if the other side still wants to make DSP calls.

The 'inndsp.jse' sample, described fully in Appendix ii, uses a peer model, so it will be instructive to look at it to see what can be done.

Using DSP

We've covered a lot of ground, from setting up distributed scripting to the models you will use. Now we get to the good stuff, actually using DSP in your scripts. Fortunately, this section is very short, since DSP scripting is very easy. Once you have a connection established with the remote host, that connection is treated very much like the entire scripting environment on the other side. We have, in our previous code fragments, named the connection variable 'conn', we will use that in the following examples.

So what does 'scripting environment' mean. Simply, all the variables and functions on the remote side can be accessed using this variable. For instance, 'conn.a' will give us the value of the global variable 'a' on the remote machine. 'conn.Clib.puts('Hi there!');' will print out a message on the remote machine, and so forth. Of course, there are a few caveats we will mention shortly, but this is the basic idea. Let's look at the potential gotchas.

DSP References

The first thing to explain is that information is not transferred between machines unless it is needed. If you write,

```
var a = conn.a;
```

'a' is not actually the value of 'a' on the other side until you try to use it, such as with:

```
var b = a + 10;
```

Now, 'a's value will be fetched from the other side to add to 10. This is especially important with objects, if you write:

```
var rClib = conn.Clib;
```

perhaps with the intention of then doing:

```
rClib.printf("Hello, world.\n");
   rClib.puts("Enter your name:);
   var name = rClib.gets();
   /* etc */
```

It would be silly to try to copy the entire Clib object across the network due to this statement. Instead, whenever you reference the other side, a special object is constructed that knows how to 'fetch itself' from the remote machine when the value of the object is needed. If you were to printout the 'typeof a', it would register as a "function" (a function

is a special type of object), not whatever type the value on the other machine is.

There are two ways you can see what 'a's type really is. The first is to force the value to be copied to the local machine, using 'dspGetValue()', for instance:

```
Clib.printf(typeof (a.dspGetValue()));
```

The second, preferred way, requires no copying of data across the machines. You simply check the type on the remote machine, and just get the type string. Here is a code fragment:

```
Clib.printf(conn.eval("typeof a"));
```

By using 'conn.eval', we run a short script fragment on the remote machine. It has no trouble checking the type of 'a' there, and then the result is send back to us for printing. This may seem a more work, but if 'a' is a complex object, trying to send the whole thing across the DSP connection can be much more work.

Working With Objects

The same gotcha can manifest itself when you want to enumerate an object's members if the object is on the remote machine. Just as 'typeof' sees the placeholder object, not the actual value, so would the FOR..IN statement. If you use FOR..IN, you will see the members of the placeholder object, not the members of the remote object. So, how do you go about getting them?

Just like seeing its type, you have two solutions. Using 'dspGetValue()' is obviously the easiest sounding one, but it will actually lead to many problems. Javascript objects can often be very complex things that don't make much sense on another machine. Cyclic loops especially can cause a lot of problems when trying to copy objects. For this reason, DSP does not allow an entire object to be copied across. We are considering ways to get this to work in future versions, but for now a workaround is needed. You need to do the enumeration on the remote machine and get the results to the local machine. So, here is the code that you would like to do:

```
var a = conn.Clib;
   for( var x in a ) { ... }
```

and this is how you should write it so it will work correctly:

```
var a = conn.Clib;
   var a = conn.Clib;
   var fields = conn.eval(
      "var tmp = \"\"; for( var tmp2 in Clib ) tmp =
tmp + \",\"+ tmp2; return tmp;");
   fields =
fields.substring(1,fields.length).split(",");
   for( var x2 in fields )
   {
      var x = fields[x];
      ...
   }
```

(Yes, this is ugly, but it works.)

DSP and The ScriptEase API

It is useful to do DSP processing via the ScriptEase ISDK API rather than via scripts in some applications. A application may want to run a server in the background and not devote an entire thread to running a server script. The application can do the same actions via the API as a script would. The source file "dspapi.c" and corresponding header "dspapi.h" provide the routines we examine.

The basic tasks for a ScriptEase API DSP program are the same as for a script; create a DSP connection, run code that uses DSP and/or service incoming DSP requests, and close everything down when done. Each task can be done by simply calling the corresponding DSP wrapper functions, finding them with jseGetMember() and calling them with jseCallFunction(). However, you can call them directly in C using the functions we will talk about next.

First, creating a DSP connection is necessary. A DSP connection is just a jseVariable that is set up with the connection information. In your case, you will need to set up the physical connection with the remote machine, then call the 'dspCreateConnection()' function, passing it two callback parameters. These parameters are C functions you provide to transmit and receive data. The resulting jseVariable can be saved, or assigned to a member of the global object if you want scripts to be able to DSP using it. In addition, this function takes parameters to allow you to set the security of this script (security is described in appendix ii.)

Now that you have a DSP object, you can perform DSP actions on it just like a script can. For instance, you can use 'jseGetMember()' to look up a member of the object, which is the remote value. Underneath, this will eventually call the required DSP routines to service the request. The function 'dspService()' is provided as well. This is a C-callable routine that acts just like the script version, allowing you to call it every so often to handle incoming DSP requests on a particular DSP connection.

Finally, you can close a DSP connection, just like in Javascript, by using the 'dspClose()' function provided.

Writing Your Own Transport

Making your own transport mechanism is pretty simply. You will need to define a new object class that handles DSP creation and transport. Doing this requires a constructor method and a couple of extra methods to handle reading data, writing data, and closing the connection.

First, you will need a constructor function. This function will create and return a new DSP connection.

Such a function looks something like this:

```
function myDSP(args)
   {
      var ret = new DSP(myOpen,args);
      if( ret!=null ) ret._prototype =
myDSP.prototype;
      return ret;
   }
```

'myOpen' is another function you will write which we talk about shortly. Arguments are whatever arguments you need to pass to the constructor. For instance, in iDSP, a client needs the host name and port number to connect to. You can pass more than one argument. We call the DSP() constructor with our open function, passing it these arguments. The result is an object of class 'DSP', but we really want it to be of our class (a subclass of DSP), and the second line accomplishes this. If you want to know more about prototypes and _prototypes, see the ISDK manual. For our purposes, it is enough to know that this is the way you do it. Finally, we return the resulting object.

The 'myOpen' function (or whatever you name it) simply takes the arguments and creates a connection to the remote host. A variable is returned representing the connection, of whatever format you choose. Often it is an object whose members store data you need to access the connection. This variable will be passed to all of the routines we will describe next. First, though, because we have changed the objects returned to be of our class, we need to make sure they still will be DSP objects. We do this by making our class an explicit subclass of DSP. You do this with the following code snippet (building on the last example):

```
myDSP.prototype._prototype = DSP.prototype;
```

Now that we have our connection established, DSP will be using it to pass its data back and forth. You need to write the necessary transport functions to allow this to happen. These functions must be put in the prototype of the constructor function so that DSP objects created with that constructor will have them. These functions only apply to DSP objects constructed with your function; you may have other DSP objects constructed with a different transport protocol, and they will use their own functions.

Building on the above example:

```
function myDSP.prototype.dspSend(conn,buffer,timeout)
   {
      /* tranport the data */
   }

   function
myDSP.prototype.dspReceive(conn,&buffer,length,timeou
t)
   {
      /* receive the data into buffer */
   }
```

Both functions take a connection as their first parameter, this being whatever value your DSP open function returned. They both also receive a timeout, the number of milliseconds to try to send or receive the data before giving up. You can ignore this parameter and just wait forever if you like. Both functions also receive a buffer parameter (a variable of type "buffer", a ScriptEase data type.)

For the send function, the buffer contains the data to be send, while for the receive function, it is a parameter you fill in with the data actually received. Remember, you can access the '.length' member of the buffer to determine its actual size.

The receive function receives one additional parameter, the length. This is the maximum size the buffer should be on return, it can be less if you receive some data but not as much as specified. Both functions return the actual length of the data transmitted.

We are almost done, one more function completes the minimal DSP transport requirements. You need a function to close down the connection when it is done. This will be called by DSP, usually as a direct result of the script using dspClose() on the connection, but it can also happen if there is an error that forces the connection to be aborted. The function looks like this:

```
function myDSP.prototype.dspCloseConnection(conn)
   {
      /* close the connection */
   }
```

Security and DSP

In an ideal world, we would be able to open up our machine to anyone in the world to connect to us and run scripts on our machines. We would hope that they would only run appropriate ones, never malicious ones. Unfortunately, that is not our world. It is necessary to implement security measures to make sure that distributed scripting is not misused.

The first part of DSP security is in the connection mechanism itself. When you are

opening a connection, you can implement a myriad of ways to verify the connector, such as by a password. However, this is usually not enough. If you are writing a chat program for instance, you want anyone to be able to connect. You just want them to only be able to do 'chat' things, but arbitrary functions such as Clib.system(). DSP allows you to do just that by adding security to the code run by remote scripts. You should take the time to read the ScriptEase security manual, since DSP security uses the normal ScriptEase security mechanism.

DSP security is implemented by adding 'dspSecurityInit()', 'dspSecurityTerm()', and 'dspSecurityGuard()' functions to the transport class, just like the 'dspSend()' and 'dspReceive()' we talked about above. These functions correspond exactly to the normal security functions, except they apply only to remote scripts run on a connection of this class. The security variable is a newly created object, but it has the "dsp" member set to the DSP connection variable.

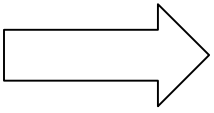
You should make sure to make all of your transport, security, and exported functions read-only. You do this using the 'setAttributes()' function, such as by:

```
setAttributes(myDSP.prototype.dspSecurityInit,0x06);
```

If you fail to do this, a clever hacker can on the first pass, change your function to one of his own, then connect again and bypass security. NEVER allow access to the 'setAttributes()' function, or your security will be able to be bypassed.

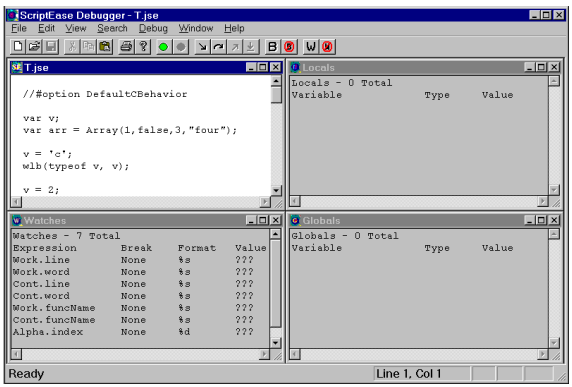
THE CHAT EXAMPLE

The programs 'inn.jse, inncli.jse', and 'innbard.jse' are long time ScriptEase samples that demonstrate a simple application using the socket library. To demonstrate most of the DSP concepts, 'inndsp.jse' is a version of all three programs that communicates by using DSP. It still uses sockets for its basic transport. This program uses a peer-to-peer model. You should look at the sample to see most of the DSP concepts described in this chapter in action.



Using the Integrated Debugger

ScriptEase comes with a source debugger that provides a complete Integrated Debugging Environment, which means you can edit a script while you are debugging it.



The debugger is a Windows application with a standard Multiple Document Interface (MDI) like many other applications. The image above has four windows showing: the script, Watches, Locals, and the Globals window. The specifics about windows are explained later. The script window is explained in the section about the File menu options, and the other three in the section about the window menu options. For now, just understand that the tiled arrangement shown above is just one out of many ways to display windows in the debugger. You may have multiple script window or only one. You may have only one window showing or any combination of windows. Like any MDI application, you may maximize, minimize, tile, and cascade windows. In short, the user interface of the ScriptEase debugger is a standard windows interface.

> ScriptEase debuggers are available only for Windows operating environments. There are debuggers for Windows 95/98, Windows NT, and Windows 3.x.

Using the ScriptEase Debugger

The ScriptEase debugger is a source code debugger, which means that you may debug programs while watching the execution of a program line by line in the original source code. You may set breakpoints, trace lines of code as they execute, step into and over functions, watch variables that you choose, keep up with global and local variables, and other powerful options that you expect in a good source code debugger.

The main window of the ScriptEase debugger consists of the following components, listed in top to bottom order.

Components of main MDI window

Menu bar

All commands in the ScriptEase debugger may be accessed through menus. The menu bar is described completely in the following section, "Main menu bar."

Tool bar

The toolbar has buttons for the common and useful debugger commands. Instead of clicking menus, you may click a button on the toolbar as a shortcut. The commands that are available on the toolbar are exactly the same as the corresponding commands in menus. In the section, "Main menu bar," commands that are available on the toolbar are indicated by the notation: "In toolbar."

Document window

The document window is a standard Windows Multiple Document Interface (MDI) window. You may open four kinds of windows within the document window: Source, Watches, Locals, and Globals.

Status bar

The status bar at the bottom of the window provides useful information concerning the currently active window. The current cursor position in a script window is displayed as line and column numbers. The status of the Caps, Num, and Scroll lock keys is displayed. When the mouse cursor is over menu and toolbar items, help or hint information displays in the status bar. The general state of the IDE is also displayed, such as "Ready" or "Program Terminated."

MDI windows

Source

Source windows may be called script windows since they display the source code of a script file. These script windows are actually text editing windows in which scripts may be viewed, edited, or used for source line debugging.

When used for editing, the editor is capable of writing an entire script, but the editing features of a script window are basic and adequate for simple scripts. Normally, you will use a more powerful editor for most writing and editing of sophisticated scripts, an editor such as the ScriptEase Editor that accompanies ScriptEase products. The ScriptEase Editor has features that allow you to coordinate your work effectively with the ScriptEase debugger. Currently, when you change text in a script while it is still loaded in a script window in the debugger, you must manually reload the file in the debugger. However, when you make changes in a script while in a script window, the ScriptEase Editor can automatically detect the changes and reload the file. Thus, for most editing of scripts use the ScriptEase Editor for major writing and script windows in the debugger for minor changes while debugging a script.

The current position in a source file is indicated by a special marker, icon, that can be chosen from several options. In addition, breakpoints may be set in a script window. Breakpoints display as small red hexagons at the beginning of the lines of scripts to which they apply.

You may open multiple script windows at the same time. Remember, that various debugging commands apply to the currently active script window. For example, a command such as "Debug | Run in Debugger" runs the script in the currently active source window, not any other scripts that might be open in source windows.

Source windows have gray backgrounds when in debugging, as opposed to editing, mode. You may not edit scripts while in debugging mode. When script windows have gray backgrounds, remember that you may only use debugging commands, such as "Debug | Step Into."

Globals

The Globals window displays all global variables that are available to the point in a script. The source marker indicates in a script where execution is currently occurring. The information for each variable displayed is the variable name, type, and value.

Locals

The Locals window displays all local variables that are available at the point in a script where execution is occurring. The source marker indicates in a script where execution is currently occurring. The variables in a local window constantly change as functions that have local variables are entered and debugged. The information for each variable displayed is the variable name, type, and value.

Watches

The Watches window is a place where you can view variables and expressions that you want to see. You may put plain variables here, and when they are active, these variables will show as in other windows. In addition you may set variables to be watched and used as breakpoints. You may set execution to break if a variable changes or is equal to true or false. But the watch window may be used with more than just variables, it may be used with expressions. For example,

the following code:

```
var arr = Array(false,1, 2, 3, "four");
```

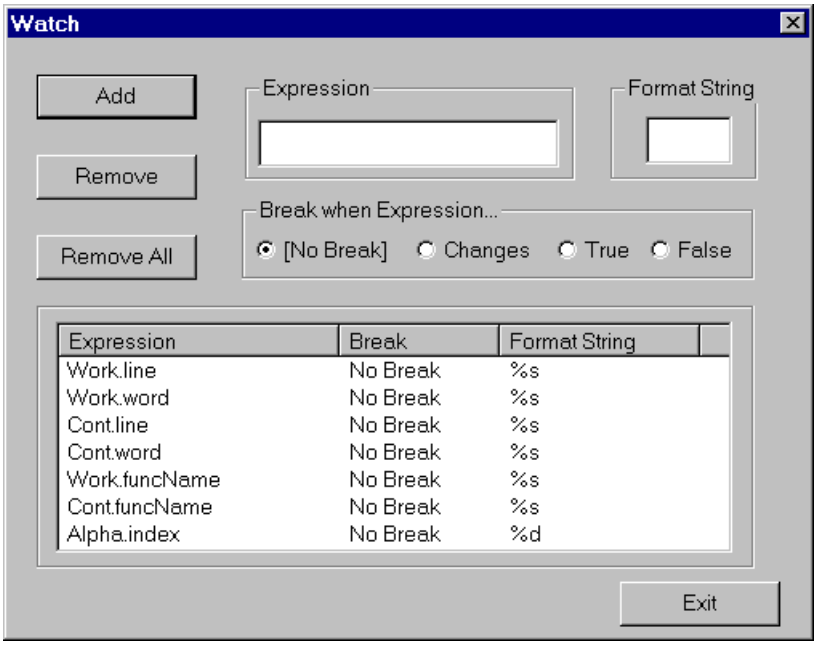
creates an array with four elements. In the Locals and Globals windows, the array `arr` is shown as type object with no value shown.

You might want to keep up with one or more elements in the array. To keep up with the second element in the array `arr`, set a watch for `arr[1]` and it will appear as an expression to be watched with its format type and value, which in this case is 1. Perhaps you want to keep up with the addition or concatenation of the fourth and fifth elements. If so, set a watch or `arr[3] + arr[4]`, which in this case would display a value of "four3".

In fact, the watch window is designed to watch expressions rather than variables. When a variable by itself is watched, the debugger simply considers it to be an expression. Notice that the second column in the watch window provides format information instead of the type of a variable.

Setting watches

The Watch dialog, Figure 2, is the main window used to set watch information.



Add

The Add button adds the current expression, in the Expression edit box, to the list of expressions to be watched in the Watches window.

Remove

The Remove button removes the expression which is currently highlighted in the list of expressions to be watched.

Remove All

The Remove All button removes all expressions to be watched.

Expression

The Expression edit box allows entry of expressions and variables to be watched in the Watches window.

Format String

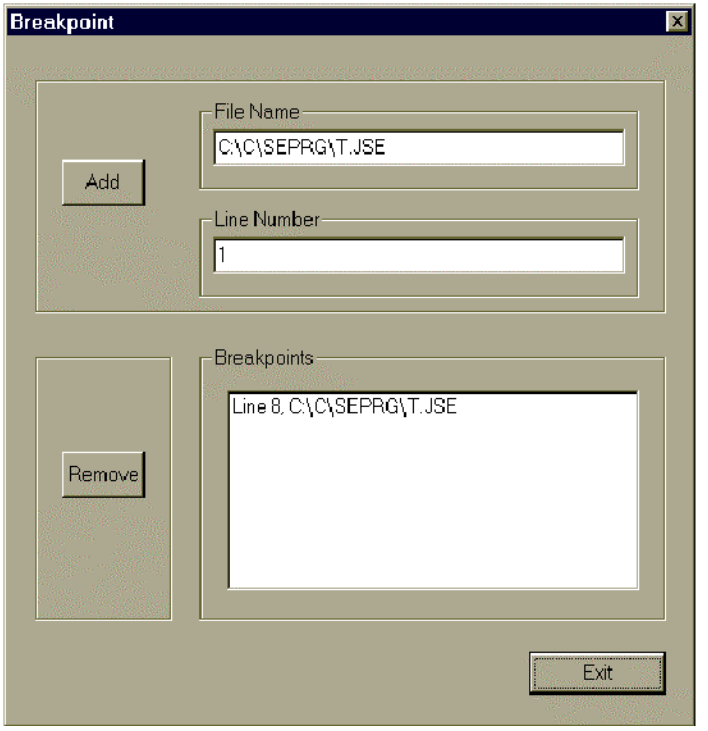
The Format String edit box allows some control over the format of expression, that is, how an expression value will appear.

Break when Expression

The four options in this group allow watches to serve as conditional breakpoints. To simply watch an expression or variable, set [No Break], which is the default. Set Changes if you want program execution to pause when the expression or variable changes value. Set True or False if you want program execution to pause when an expression becomes true or false. You may use "Debug | Change Variables..." to set a variable to a different value and watch execution with the changed variable.

Setting breakpoints

The Breakpoint dialog, Figure 3, is the main window used to set breakpoints.



Add

The Add button adds a breakpoint at the line specified, in the Line Number edit box, to the script specified in the File Name edit box. Of course, the script itself is not altered since scripts are plain text files. Breakpoints are retained as settings within the ScriptEase debugger.

Remove

The Remove button removes the breakpoint which is currently highlighted in the Breakpoints list box.

File Name

The File Name edit box indicates which script is presently being used for add and remove operations

Line Number

The Line Number edit box indicates which line in a script is affected by add and remove operations

Breakpoints

The Breakpoints list box shows all breakpoints currently active in a script.

Main menu bar

The main menu bar consists of the seven menus across the top of the windows just below the title of bar. The seven menus are: File, Edit, View, Search, Debug, Window, and Help. Some menu commands may be accessed from the toolbar or by shortcut keys, and those that can are indicated by the notations: "In toolbar" and a keystroke description.

File menu

The file menu has options for starting, opening, closing, saving, and printing script files. Plus, an exit option to exit the debugger. All of the commands concerning files operate on script or source files. These files are opened in the integrated editor which allows the use of all debugging options in the integrated debugger. The editor is also a standard editor that can be used to do plain text editing in any text file, such as one created by Notepad.

The editor can be used to write complete scripts. Normally, however, scripters use their favored editors to write and edit most scripts and use the integrated editor while debugging a script.

New In toolbar and Ctrl+N

Start a new script or source file. The file is opened in the editor which is integrated with all debugging features.

Open... In toolbar and Ctrl+O

Open dialog to open a script file.

Close Ctrl+W

Close the currently active script file.

Save In toolbar and Ctrl+S

Save the currently active script file.

Save As...

Save the currently active script file to a new filename. The title of the currently active script will change to the new filename. Immediately after a script is saved to a new filename, the script will exist in two separate files with the old and new filenames. But, the new filename will be the active script. To edit the previous file, it must be opened again.

Print... In toolbar and Ctrl+P

Print the currently active script file using straightforward print settings. The print dialog that opens is a standard Windows print dialog.

Print Preview

Preview how the printed script file will look before actually printing the file. When previewing a page, there are various options to page through the pre-printed document, examine pages one or two at a time, zoom in and out, print the document, or close the preview window without printing.

Print Setup...

Change printer settings. These settings are for the printer and are not a page setup. The print setup dialog that opens is a standard Windows print dialog.

(Recent files list)

List up to four of the most recent script files that have been opened in the editor.

Exit

Exit the entire ScriptEase debugger program. Some settings, such as the size and location of open windows is saved. Thus, when the ScriptEase debugger is started again, it is easier to restore various windows to their previous state.

Edit menu

Undo Ctrl+Z

Undo the last editor operation in the script window.

Cut In toolbar and Ctrl+X

Cut selected text from the script window.

Copy In toolbar and Ctrl+C

Copy selected text from the script window.

Paste In toolbar and Ctrl+V

Paste text at the insertion point, where the cursor is, or into the selection in the script window.

Options

Font...

Display a dialog to set the style, size, and color of the font used in the debugger windows.

Tabs...

Set how many spaces should be used when displaying a tab character in the debugger windows.

Trace On

When a script is run using the Debug | Run in Debugger menu item, the active script runs until it encounters a breakpoint or the script ends. If the Edit | Options | Trace On option is checked, then when a script is run in the debugger, the lines executed are traced. The source marker visibly moves from source line to source line as the script is run. The effect is similar to choosing the Debug | Step Into and Step Over menu items. The difference is that with Trace On checked, the stepping is done automatically.

Trace Speed

When the Trace On menu item is checked, the Trace Speed options determine how fast the trace operation executes each line of a script. The options are: Fast, Normal, Slow, and Slowest.

Trace over

When the Trace On menu item is checked, the Trace Over menu item determines if the tracing steps over functions that are called or steps into them. When Trace Over is checked, the tracer steps over functions, and when it is not checked, the tracer steps into functions.

Source Mark

When debugging a script, the current position in a script is visibly marked by an icon or graphic. The Source Mark option allows a choice of the appearance of the marker.

Default Interpreter...

The default interpreter is the ScriptEase executable that the debugger uses when executing a script. In Win32, the two valid programs are SEwin32.exe and SEcon32.exe. There are differences between a windowed application and a console application. You

may want to set the default interpreter to be the same interpreter that you will use to execute a script.

View menu

Toolbar

View the push button toolbar, just below the menu bar, if checked.

Status Bar

View the status bar at the bottom of the debugger window. The status bar displays various helpful messages and the position of the cursor or insertion point in the editor in terms of line and column.

Search menu

Find...Ctrl+F

Find text in the script window using a find dialog.

Replace... Ctrl+R

Find text in the script window and replace it with other text using a find and replace dialog.

Debug menu

Start Debug Session

Start executing the active script in a debug session. The source marker is positioned at the first executable line in the script awaiting further commands.

Restart

Restart a debugging session. The source marker is positioned at the first executable line in the script awaiting further commands.

Run in Debugger In toolbar and F5

Run the current script in the debugger. The source mark appears. The script executes until a break point is reached or the script is finished.

Go Ctrl+F5

Execute the current script as a program, that is, not in the debugger.

Stop In toolbar

Stop the execution of a script that is running in the debugger. The script may be actively executing or paused at a source line or breakpoint.

Step Into In toolbar and F9

Steps into any user defined functions in the current source line and begins displaying source lines in the function as they are executed. Does not step into built in functions. If a script has not begun execution in the debugger, then the first line of executable code is executed.

Step Over In toolbar and F10

Steps over any user defined functions in the current source line and simply executes the line and pauses at the next line in the current script. If a script has not begun execution in the debugger, then the first line of executable code is executed.

Step to Cursor In toolbar and F11

Executes all lines of executable code till reaching the line where the cursor is located. In effect, the cursor behaves like a temporary breakpoint.

Step Out In toolbar and F12

Executes lines of code in the current function until the function is finished.

Parameters...

Opens a dialog box to set command line parameters to be sent to a script when it is executed in the debugger. The parameters are handled by a script in the same way as they are when part of a command line.

Breakpoint

Toggle current In toolbar and F8

Toggle the breakpoint at the current line, off or on.

Add/Remove...

Opens a dialog box to add or remove breakpoints on any line in the current script.

Remove all In toolbar

Removes all breakpoints in the current script.

Watch

Add/Remove... In toolbar

Opens a dialog box for adding variables and expressions to the watch window or removing them.

Remove all In toolbar

Remove all watches from the current script and debugging session.

Change Variables

The menu item allows a variable to be changed while a script is executing.

Window menu

Cascade

Display the open windows in the debugger in a cascaded fashion.

Tile

Tile open windows horizontally. If two or three windows are open, they are all tiled horizontally extending the entire width of the main debugger window. If four or more windows are open, then two columns of windows are begun, and all windows are tiled horizontally in the two columns. For example, if a script window, the global, the local, and the watch window are opened, the resulting window is quartered. Each window will be in the four corners of the main window. The screen shot, Figure 1, at the beginning of this section is an example of four tiled windows.

Arrange Icons

As in all MDI applications, open windows may be minimized inside the main window. The Arrange Icons menu item arranges these minimized icons at the bottom of the main debugger window.

Global... Ctrl+Shft+G

Open the Globals window to view global variables while debugging a script.

Local... Ctrl+Shft+L

Open the Locals window to view local variables while debugging a script.

Watch... Ctrl+Shft+W

Open the Watches window to view variables and expressions that have been defined by a user.

(Open windows list)

A list of the currently open windows in the debugger.

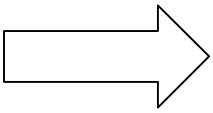
Help menu

Help Topics... F1

Display a help file for the debugger.

About ScriptEase Debugger... In toolbar

Displays program information, version number, and copyright notice for the debugger.



Appendix I

Aboutopt.jse - jseopt.h analyzer

Purpose

The jseopt.h ananlyzer is designed to provide the user with three important pieces of information. The first is which files the user needs to include in his or her project in order to have all the functions available. The second is the functions which need to be called in order to make the functions available scripts. The final information is exactly which objects and functions will be made available, based on what the user has defined. The analyzer is really a C preprocessor which analyzes what has been left defined after processing the file in order to see what information the user needs.

Configuration File

The aboutopt.cfg file is used to manage additional define and user-specific search directories. The script searches for this file both in the current directory and the directory where the script is located. The config file is actually just a script that is executed by the program, so you have all the control that you would normally have in SEDesk. There are two things that the config file is designed to do:

Define constants

Many times you need to have certain defines set to correctly process a file. For example, you may need to have __JSE_WIN32__ set to include all the appropriate files. While these defines can be specified on the command line, you have more control within the config file (such as being able to define each to something other than 1). To do this, simply set the global value and what you want it to be defined to be.

This section of a config file may look something like this:

```
__CENVI__ = 1;
NDEBUG = 1;
if defined(__JSE_WIN32__)
{
__WATCOMC__ = 1;
__WINDOWS__ = 1;
}
```

This section defines some globals for general use, which are __CENVI__ and NDEBUG. And since this is an interpreted script, it can use the control statements to define additional defines if __JSE_WIN32__ is defined. This particular config file will 'fill-in'

the minor defines (the compiler and system defines __WATCOMC__ and __WINDOWS__) given a 'major' define (__JSE_WIN32__).

Designate search paths

During the analysis process, the script must be able to find include files. The most important of these files are the user include paths. You must make sure that you include 'incjse', 'srcmisc', 'srcapp', and 'srclib' in your search directories. You may also include system paths, but for the most part these should be unecessary. The only thing that include files are scanned for are additional defines, so the only reason you would need to include one of these files is if there is a conditional define that you need, although you could always define it yourself in the config file.

There are two functions you can use to add user paths to the search paths. They are `AddUserPath()` and `AddSystemPath()`. They both take as their first parameter the path to the directory to search, which can be either a full path or a relative path. The second optional parameter, if true, indicates that the directory should be searched recursively. This is important in such directories as 'srclib', where you don't want to have to specify every separate lib directory.

This section of a config file might look something like this:

```
var ISDKPath = Clib.getcwd() + `\`;
AddUserPath( ISDKPath + "incjse" );
AddUserPath( ISDKPath + "SRCMISC" );
AddUserPath( ISDKPath + "srcapp" );
AddUserPath( ISDKPath + "srclib", true);
```

Again, because it is an actual interpreted script, the config file can use functions such as `Clib.getcwd()`. This section of code assumes that you are in the root directory of the ISDK tree, and builds the paths from there. Note also that `true` is passed as a second parameter in the last call to the function, which indicates that 'srclib' should be searched recursively.

Usage

Once the configuration file is specified, the program simply needs to be run by running `secon32 aboutopt`. The program will attempt to look for the configuration file in the current directory, or in the directory in which the script is located. Additional defines can be passed to the program by specifying parameters to the script. Any additional parameters passed to the script will be defined to be 1 during analysis of the header file. For example, running `secon32 aboutopt __JSE_WIN32__` will define __JSE_WIN32__ to be 1, just as if it was set in the configuration file. For the most part,

all defines should be put in the configuration file, although if the header file is used for multiple OSes, it might be better to allow the OS to be specified on the command line.

Because the configuration file is really just a script, and the command line options are initalized before running the configuration file, a config file could have text such as:

```
if defined(__JSE_WIN32__)
{
__WATCOMC__ = 1;
__WINDOWS__ = 1;
}
```

This defines some additional parameters, but only if __JSE_WIN32__ is supplied on the command line. The output is sent to standard output, although most likely you will want to redirect the output to a file. The first messages are simply a list of files as they are processed, and errors that came up during that process. The important part of the output is after this list, and is described below.

Understanding the Output

The output from the analyzation process is divided into four basic sections:

Files you need to include

This section lists the files which you should include in your project in order for all the functions you have specified to be included with the interpreter. If you fail to include all of these files, most likely you will get linking errors and unersolved references. This section is straightforward, with the files being divided into groups.

The output often looks something like this:

```
Files that you need to include:
/**** SRCMISC ****/
srcmisc\utilhuge.c
srcmisc\sedllrun.c
srcmisc\globldat.c
srcmisc\seobjfun.c
srcmisc\cmdline.c
srcmisc\dbgprntf.c
srcmisc\dirparts.c
srcmisc\jsemem.c
srcmisc\findfile.c
srcmisc\utilstr.c
/**** SRCLIB\COMMON ****/
srclib\common\setxtlib.c
srclib\common\sedyna.c
srclib\common\sedynlib.c
srclib\common\sedyncal.c
srclib\common\seliball.c
srclib\common\selink.c
srclib\common\selibutl.c
srclib\common\seblob.c
/**** SRCLIB\LANG ****/
srclib\lang\selang.c
srclib\lang\selngmsc.c
srclib\lang\seconvrt.c
/**** SRCLIB\ECMA ****/
srclib\ecma\mathobj.c
srclib\ecma\seecma.c
srclib\ecma\ctables.c
srclib\ecma\regex.c
srclib\ecma\sebuffer.c
srclib\ecma\seregexp.c
srclib\ecma\ecmamisc.c
srclib\ecma\sedate.c
srclib\ecma\seobject.c
```

Simply include all of the specified files into your project and you'll be set.

Functions you need to call

This section describes all of the functions you will need to call in order for the functions to be added to the interpreter. This allows for individual enabling of libraries, though in the majority of cases, you should simply call `LoadLibrary_All()`, which will call all the appropriate load functions.

The output for this section looks something like the following:

```
Functions you need to call to initialize libraries:
LoadLibrary_Lang();
LoadLibrary_Ecma();
```

Alternately, you may call `LoadLibrary_All()` which will call all of the above functions.

All of these functions take a single parameter, which is a `jseContext`. The context returned from `jseInitializeExternalLink()` should be passed to these functions during the program's initialization. Any contexts inherited from this base context will include the libraries added to the original.

Objects available to scripts

This section describes which global objects are available to scripts, and which methods instances of those objects have available to them. These objects can be called as constructors, such as `new Array()`, and the instance variable that it returns will ahve all of the specified functions available to it. These functions are equivalent to `Array.prototype.function`, as they are inherited through the prototype chain. The output for this section looks something like the following:

```
Object constructors available to scripts and methods
of those objects:
Array
.concat()
.join()
.pop()
.push()
.reverse()
.shift()
.slice()
.sort()
.splice()
.toLocaleString()
.toString()
.unshift()
Boolean
.toString()
.valueOf()
Buffer
.putValue()
.getValue()
.putString()
.getString()
.toString()
.subBuffer()
Function
.apply()
.call()
.toString()
Number
.toLocaleString()
.toString()
.valueOf()
Object
.hasOwnProperty()
.isPrototypeOf()
.propertyIsEnumerable()
.toLocaleString()
.toString()
.valueOf()
String
.charAt()
.charCodeAt()
.concat()
.indexOf()
.lastIndexOf()
```

```
.localeCompare()
.slice()
.split()
.substring()
.toLocaleLowerCase()
.toLocaleString()
.toLocaleUpperCase()
.toLowerCase()
.toString()
.toUpperCase()
.valueOf()
```

The output is organized into sections by object. The methods listed under each section are defined as members of the prototype member, and every instance has these methods available. For example, in the above example, a script can call, `"var string = new String(); string.concat(4)"`.

Functions available to scripts

This section is very similar to the above section. The only difference is that the objects are not constructors, and cannot be called as functions. This section is simply a list of all functions available to scripts. Some of these functions may be grouped into objects (as with the SElib and Clib objects), although instances of these objects cannot be created as in the above example. The output of this section looks something like the following:

```
Functions available to scripts:
Blob
.get()
.put()
.size()
defined()
escape()
eval()
getArrayLength()
isFinite()
isNaN()
Math
.abs()
.acos()
.asin()
.atan()
.atan2()
.ceil()
.cos()
.E
.exp()
.floor()
.LN10
.LN2
.log()
.LOG10E
.LOG2E
.max()
.min()
.PI
.pow()
.random()
.round()
.sin()
.sqrt()
.SQRT1_2
.SQRT2
.tan()
Number
.MAX_VALUE
.MIN_VALUE
.NaN
.NEGATIVE_INFINITY
.POSITIVE_INFINITY
parseFloat()
parseInt()
```

```
SElib
.directory()
.dynamicLink()
.fullpath()
.getObjectProperties()
.inSecurity()
.interpret()
.interpretInNewThread()
.peek()
.poke()
.spawn()
.splitFilename()
.suspend()
setArrayLength()
String
.fromCharCode()
undefine()
unescape()
Win
.asm()
.windowList()
```

Note that both properties (i.e. Number.MAX_VALUE) and functions are represented in this list. It is a complete list of everything made available to scripts, besides the functions inherited through objects. For example, the `.directory()` item means that the user can call the function `SElib.directory()`. Functions which are not part of an object are simply called directly.

Appendix II

Under the Hood

- Advanced Topics

Topic 1: What is a jseContext?

A number of users seem confused over what exactly a jseContext is. A jseContext is an internal data structure that the API user passes to all API functions. It is a magic cookie to the API user. The structure is actually a linked-list, so the pointer value can change over time; this is why your wrapper functions are passed a jseContext, and they must use that value when they make API calls. You cannot test jseContexts against one another by direct pointer comparison.

Each jseContext can only be used by one thread at a time. If multiple threads need to be running scripts simulataneously, each needs to create its own jseContext (using jseInitializeExternalLink.) It is not enough to prevent task switching to try to overuse a single context. The jseContext contains state information that cannot be intermixed. However, you can successfully share jseContexts among tasks if only one thread uses it at one time. For instance, a thread may grab a jseContext and do a jseInterpret() using it, then when it is finished, give it to some other thread to do its own jseInterpret(). If both try to do a jseInterpret() simulataneously you will crash.

You can associate data with a jseContext and then retrieve the data using jseGetLinkData(). This allows you to differentiate contexts, even though you cannot directly compare pointers.

Topic 2: Errors and Exceptions

Another point that needs clarifying is exception handling. An error and an exception are the same thing. A ScriptEase wrapper function can generate an error using the jseLibErrorPrintf() function. It is preferable to use the resource capabilities to generate the error message, since then you do not have to worry about the format of the error string. All of the standard library functions we provide have their error messages generated this way, such as the standard ECMA functions in srclib/ecma/*.c. The file srclib/common/setxtlib.h contains the resource definitions they use.

If you need to generate an error message by hand, the format of the string passed to jseLibErrorPrintf is:

```
!type number: message
```

For instance,

```
!SyntaxError 1000: You did something wrong.
```

Would generate an error of type 'SyntaxError' (one of a number of error types defined by the ECMAScript Version 3 draft document), with number 1000 and the message 'You did something wrong.' You can pass the string without the type specifier portion, in which case you will get a generic error rather than a specific one (i.e. instead of being a TypeError or a SyntaxError, it will just be an Error).

In Javascript, you can generate errors at runtime by 'throw'ing them. The equivelent of the above error would be the statement:

```
throw new SyntaxError("1000: You did something
wrong.");
```

Normally, errors will cause the program to stop execution and print an the error message. You can trap errors. In Javascript, you use the try/catch mechanism (see the manual for details). The API allows you to trap errors that result from calls to jseInterpret() or jseCallFunctionEx(). See the API references for how you do that.

The rule is straightforward: if you do not trap the error, the error is printed using your error print routine defined in the jseExternalLinkParameters structure, and the function returns failure. If you trap it, the function still notifies you of its failure, but the Error object is returned as the result of the function. Normally, you wouldn't then print the error, but you can do so by converting the error object to a string using the jseCreateConvertedVariable() API call with the 'jseToString' conversion option.

Now that we now the basic mechanism, here a couple of tidbits of information you may find interesting and helpful. First, you can throw any value, but normally you will want to throw some kind of error object. In the handler, when you catch an error of some type, the associated error object will be assigned to the variable you specify in the catch handler. If you throw an error object, the variable will be an error object. If you throw something else, the person's handler might get confused if it was expecting some kind of error object. Exactly the same applies to the API, you may get a failed call, but the return value is not an error object. In Javascript, you are allowed to have any value associated with an error condition; you just normally make it an error object of some kind.

In wrapper functions, it is common to want to pass an error back up the chain. For instance, the ECMA eval function wants any errors that it gets to be applied to the calling

context. There is no trick to doing this. Call whatever function you want to trap the error. Then, if you get an error, you can pass it up the chain by using jseReturnVar() on it. You'll call jseLibSetErrorFlag() after to indicate that the return was an error. Exactly, like we talked about above, we return some value and indicate that it is an error result. jseLibErrorPrintf() is just a convenient function to handle the usual case of generating a stock error from the ScriptEase API.

Topic 3: Jseinterpret And Scripts, Functions, Variables, And Scoping

In order to support object-oriented programming, classes, executing scripts as children of other scripts, and similar scoping issues, the options available to run scripts in the ScriptEase: ISDK API can be confusing. This chapter will try to sort out the issues involved.

Let's start with a stroll down memory-lane. Bear with me, it really is relevent! If you recall your schooling days, in math class often you were expected to learn some formulas. You plugged in the right numbers and got the answer. They seemed mysterious and magical. But later, as you began to understand where the formulas came from, they didn't seem so magical. When you got a problem that was different enough that no formula worked exactly, you understood what was going on and could figure out how to use what you did know to solve the problem.

The ScriptEase engine is exactly the same way. You can think of this chapter as giving you some 'formulas' for the common tasks associated with loading and executing scripts and determining variable scoping. However, it also explains how these 'formulas' work so if you need to do something that isn't covered here exactly, you can figure out how to do it. I encourage you to read this document several times to make sure you get it all.

Functions and the Global Variables

Let's start with the Javascript basic idea of the Global Object. The Global object is an object that is special. Every jseContext has a single global object. This is the place where global variables and functions are stored.

When you run a script using jseInterpret, all of its functions are parsed and stored as members of the global object. It also actually runs the script code, we will get into that part later. Remember, functions are just variables that happen to be able to be called as functions. Since these functions are stored in the global variable, they overwrite any functions or variables that are already there. Even after the script is finished executing, the functions are left over. This leads us to point 1:

Point One:

When a script is interpreted using jseInterpret(), any functions it defines remain in the global object after it finishes.

This is a point that often confuses people. Some people mistakenly think that jseInterpInit() is needed to get at the functions defined in a script. This is not true. jseInterpInit() is only provided so that you can execute your script one statement at a time under your program's control. It's use is probably unneeded for 99% of customers, as the MayIContinue function (defined in the jseExternalLinkParameters structure) can do what you need instead. In fact, there is NO functionality related to scoping that jseInterpInit() has that cannot be done using jseInterpret(). If you are having problems getting your functions and variables correctly visible, then your problem is not related to using the 'wrong' function. jseInterpret() is the correct one.

Point Two:

The difference between jseInterpret() and jseInterpInit() is only important for controlling execution, not for affecting scoping.

If you need to run a script a statement at a time, perhaps only occassionally when your application has time, or if you need to run multiple scripts simultaneously in a single thread intermixing them, this is a job for jseInterpInit(). Scoping is not.

Interpreting Children

As I said, the functions and variables defined a script used by jseInterpret() hang around when they are done. This is the default behavior of jseInterpret(). However, there are other values you can pass in order to get different behavior. The ScriptEase API is very flexible; no matter what behavior you want, you can get it with the right combination of API calls and parameters.

You can think of a 'stock' call to jseInterpret to look like this:

```
result = jseInterpret(jsecontext,
                      "my script code",
                      NULL,
                      jseNewNone,
                      JSE_INTERPRET_DEFAULT,NULL,NULL);
```

As I talk about various alternatives, like 'adding' a flag, it is understood to be modifications of the basic format given above. There are several fields that we will not be discussing at all. If instead of giving script code, you opt to give a filename or

precompiled bytecodes, that is independent of what we are talking about. All of the scoping behavior applies the same to a script file or source code text or bytecodes. You can read the API reference on jseInterpret() to learn more about it.

The default behavior is such that if you run a series of scripts, all the functions defined in them will remain, with newer copies of functions of the same name taking precedence. This may not be what you want. Let's start with a common case. You want to run a script as a child, with its functions overwriting any existing functions, but only for that child. When the child finishes up, you want its functions are variables to go away, and everything that was there to begin with to remain.

To accomplish this, pass the NewContextSettings value of 'jseNewFunctions' to jseInterpret() instead of jseNewNone. The name of the flag is historic. Nombas is committed to backward compatibility whenever possible. That means that names of certain functions and parameters are kept the same, even if we would give them a different name today. The behavior is kept the same, or as close to what it used to do as possible. In this case, at one time functions were kept separate from regular variables, and this flag said we wanted to create a new set of functions for this interpret. That's basically still what it does, hence the historic name.

Finding Existing Variables

Remember when I said there were two conceptual parts to interpreting a script? The first is loading any functions. What about the second? The second part is to run any code in the script. For instance,

```
function foo()
{
   Clib.printf("Hi there. Please make your selection.\n");
}

foo();
```

Is really two parts, creating the function foo(), then running the little snippet of code 'foo();'. All of the statements outside of any functions is the code to run. Often, if you are using the script merely to define functions, there will be no code at all, since the point of your script is just to define those functions. Running the code is pretty straightforward, it runs whatever script you specified. Any variables the script defines will either stay or go away when it is done just like any functions it defines. But what happens when it refers to variables that it doesn't define, variables that already exist? How does that work? You get to decide. A couple of other fields to jseInterpret() determine how the child views variables that already exist. Consider the simple script:

```
var a = 10;
var b = a;
var c = d;
```

In this case, 'b' will obviously always be 10. 'a' will be 10 as well. As we just discussed, whether or not 'a' remains 10 when the script finishes depends on whether you use 'jseNewNone' or 'jseNewFunctions' in your call to jseInterpret(). But, what about 'c'? It is set to the value of 'd', but 'd' is not defined in this script. There are two possibilities, and you get to decide which it is. Assuming that the variable 'd' was already defined by the parent, either you want that 'd' to show up, or you want there to be an error because the child script does not define 'd'. You need to decide whether or not the child gets to see the variables of the parent. This is done with one of the 'HowToInterpret' flags, another parameter to jseInterpret(). The flag JSE_INTERPRET_NO_INHERIT makes all the difference here. If it is included, then any variable that existed is hidden from the child. If it is not included, the child can see the old variables.

This flag only affects global variables. There can be times when you want the child to be even more like the parent. For instance, the ECMAScript eval() function lets the child inherit everything. If the parent was in the middle of a 'with' statement, those variables are visible. If it is in a function, all of the function's locals are visible. If you want your child to have access to everything the parent did, make sure the JSE_INTERPRET_NO_INHERIT flag is not on, and pass your current context as the 'LocalVariableContext' parameter to jseInterpret(). The child script will get an exact copy of the parent's scoping chain, and thus will see exactly the same variables that its parent could.

One clarification. The context does not try to remember which variables have been changed. New variables go away because they are created in a new place. If you access an existing variable and change it, for instance by taking an existing object and adding a new member, that will not go away. New variables go away, changes to existing variables do not. I know this is a bit confusing. If the person can see one of the existing variables, they can change it.

jseInterpret Context Settings

Because any script can always see the global object and its variables, you must change the global object if you want to use JSE_INTERPRET_NO_INHERIT to completely hide all existing variables from a child. Usually, you will use the new context settings of 'jseAllNew'. Since the ECMA libraries, ScriptEase predefined constants, and such are all stored in the global object, you'll do this to initialize a new copy of them which the child can use. This is basically equivelent to just initializing a whole new context, except that you don't have to go through and respecify the libraries; the engine reinitializes all the ones you've added.

You can individually turn on each of the new context setting flags to reinitialize a particular subset of the engine for the interpreted child. This is probably not very useful and I will not get into it here. The flag jseNewSecurity is, however, quite useful. If it is turned on, the security specified in the jseExternalLinkParameters structure (accessible via the API function jseGetExternalLinkParameters) is used to add a new level of security. You'll often want to apply additional security to a certain script you want to run, and this is how you do it. See the security chapter for full details.

Mixing In One Call

Scoping for interpreting children is easy for you to figure out and implement. The child runs and then finishes, and you only need decide what happens to its variables and functions. Many applications will need a more complex situation. Browsers are a perfect example. Functions need to be associated with certain objects and the variables they can see set accordingly. Later on, all of the functions are going to be available at the same time, but each must see its own particular variables. For instance, a function associated with a form must see that form's variables, its containing document's variables, as well as the variables in the containing window. In this case, we want to set up the particular hiearchy of scoping and have it all be remembered so that whenever these functions are run, they get the correct values.

Remember POINT ONE. Any functions defined by jseInterpret() stick around after jseInterpret() goes away. This is what we want to happen, but we do not want all the functions mashed togethor in the same place. We want the functions associated with the form to go with the form, the functions associated with the window to go with the window, and so forth. If we create the functions 'window.foo()' and 'window.document.forma.foo()', those are two different functions, so we obviously don't want them overwriting each other. This is common with event handlers, as handlers with the same name will be needed for different objects in the browser tree.

Changing The Global Object

In this case, we know that jseInterpret() will leave its functions hanging around in the global object. If we want our functions to go in different places, obviously we are going to need more than one global object. In fact, the 'jseNewFunctions' flag we looked at previously just creates a new global object for the jseInterpret() and gets rid of it when it is done.

Point Three:

The Global Object Can Be Changed!

By calling jseSetGlobalObject() and then calling jseInterpret(), we can determine where the functions in that script go. If we interpret several scripts, each with their own global object, the functions will get put into their own space. In the browser case we looked at above, when we interpret the 'function foo()' that corresponds to

'window.document.forma.foo', we look up 'window.document.forma', make that the global object, then interpret the script. Then if we want to make the 'function foo()' that corresponds just to 'window.foo', we need only swap 'window' back as our global object and interpret again.

This new function goes in its new place, and the other one is also hanging around. Just remember to use 'jseNewNone', as we will be setting exactly which global object we want, so we don't want to then have jseInterpret() change it on us. The scoping is not important, since we will typically be only using jseInterpret()s capability to create functions. After the browser has set up all of its functions, it will later be calling them using jseCallFunction(), or by running more scripts.

Sophisticated Scoping

By default, the ScriptEase compile-time define 'JSE_MULTIPLE_GLOBAL' is turned on. This means whenever any function is run, the global object that was in effect when it was created becomes the global object for the function. This is usually exactly what we want. Continuing with the browser example, when the 'window.document.forma.foo()' function is run, it should see all of the variables in 'forma' as if they were its globals. When 'window.foo()' is run, it gets to see the variables in 'window'. But, you may notice, it is more complex than just that. 'window.document.forma.foo()' actually is supposed see the variables in 'window.document.forma' AND 'window.document' AND 'window'. All of the variables in the containing object and all of its parents are supposed to visible.

How can we accomplish this? Well, we have a good start so far. By swapping in the correct global, we get the 'first piece' visible. More importantly, we have made sure that if the function creates any 'global' variables, they actually get stuck into their own little place. Now, we want to make the whole chain we just discussed visible as well. The first thing we need to do is chain the global variables togethor. We know that a form has a document as its parent and that a document has a window as its parent, and so forth. We need to tell ScriptEase that. We do this by making each object have an '__parent__' property. This is done using the standard ScriptEase API.

For instance,

```
jseVariable window;   /* this is the ScriptEase variable
                         associated with a window. */
jseVariable document; /* This is a document inside the
                         above window. */

jseVariable tmp;

tmp = jseMember(jsecontext,
                document,
                "__parent__",
                jseTypeUndefined);
jseAssign(jsecontext,tmp,window);
```

In our sample browser library, you'll notice this kind of thing going on a lot. Whenever a particular object is created, it gets a number of child objects. The creation routine fetches the child object, makes it one of its members, then sets the child object's "__parent__" property to point to itself.

Great, we are almost there. The __parent__ chain is a runtime property, not a compile-time one. What that means is that when a function is actually being run, only then does it grab all the parents and make their variables visible. This is because which variables are going to be visible will be determined by the 'this' variable that the function is actually used on. In the browser example we have been doing, we can see what this value will be at compile-time.

For instance, when we compile a function the function 'forma.foo', we are associating it not with forms in general, but with the 'forma'. When we run it, such as with 'forma.foo()', the 'this' variable will obviously be 'forma'. But this is a specific case of a more general ability in ScriptEase. Thus, the mechanism that we have added allows the specific case to be done, but more general cases as well. We simply need to mark the function with some function attributes needed to make them visible. You can see the API reference for the ScriptEase function jseSetAttributes() for more information.

We want to mark the function as having the attributes 'jseImplicitThis' and 'jseImplicitParents'. jseImplicitThis means make all the varables in 'this' visible. In other words, we can reference them directly instead of having to put a 'this.' in front of them. jseImplicitParents is similar, except it allows the variables in the __parent__ to be seen in the same way. That chain of __parent__ is followed until the end, so the parent's parent and so forth are also visible. By setting up your parent chain and making your functions jseImplicitParents, you can control exactly which variables you want visible in your functions.

Observent readers will notice that 'this' and the global object are the same, so jseImplicitThis seems unnecessary. Variables in the global object are always visible. True, but the implicit this is seen before the implicit parents, while the global is seen after. Since the browser programmer will expect the 'this' values to take precedence, we include jseImplicitThis.

Again, there is no magic involved. To set the attributes, you retrieve the function you've just created using jseGetMember() and set its attributes using jseSetAttributes().

What About Prototypes?

You've probably read the earlier section in our manual about dynamic objects and prototypes and you may ask, why do I need all of this __parent__ stuff? Can't I just link my objects togethor using the prototypes instead? The short answer is yes. However, for a

browser, this is not enough. Prototypes are used to differentiate classes of objects. For instance, all documents are of the same class, have the same functions available, and thus get the same prototype object which has in it the functions for all documents.

The chain of parents, though, is different. Each document has a parent window, but not all documents get the same parent. They are part of one chain for variable visibility in which each document may have the same or a different window as its parent, while at the same time all documents are still documents so have the same prototype. If we tried to use the prototype chain alone to do both tasks, we just wouldn't get it to work. If your application and object hiearchy are very simple, you may be able to use just the prototype chain, but for most real applications, it makes sense to use each as it was intended.

Multithreading

Some people are confused about what a context is. A context is an internal data structure used to track a single thread of execution. Multiple threads cannot use the same context. If multiple threads wish to execute code simultaneously, each one will need to get its context to work with using jseInitializeExternalLink. While it is possible to share data, that is outside the scope of ScriptEase. The various data structures, such as contexts and variables, cannot be shared by threads.

This is due to the problem of multiple threads trying to access the same data. Having a semaphore to limit physical access to the underlying memory is not enough. If I am in a function that converts a variable to an string then puts data into that variable, even though each of those operations might be locked via semaphore, their combination in a sense is atomic. If some other task changes the variable to a number in between the two steps, we will get a crash.

If you must share data, my suggestion is to use dynamic _get and _put to access a shared data structure, in which the _get and _put can do the appropriate sharing, perhaps via a semaphore. Having two or more threads simultaneously accessing a ScriptEase data tree is just not a realistic possibility. It could be done, in theory, but it would be no easy task.

jseInterpret Deprecations

jseInterpret() has been a staple API call for a while, and in our effort to ensure backward compatibility, certain flags have stayed around that probably don't need to be. If you are writing a new program, you may be looking at some of these flags and wondering why they exist.

JSE_INTERPRET_LOAD - This flag used to be necessary to get the code to load into the existing context. This has been superceeded what object becomes the global variable as described above. The flag does nothing.

jseNewGlobalObject - A context setting which is very similar to jseNewFunctions. However, jseNewFunctions makes a few other changes under the hood to make sure its functionality is as described. This flag probably shouldn't be used independently. Use it only when you use jseInterpret() with 'jseAllNew' as its NewContextSettings. You are probably better off just using a whole new context via jseInitializeExternalLink().