



GoLive™ 5.0 Extend Script SDK Programmer's Guide

SDK Version 1.0
For Adobe® GoLive Version 5.0

This book describes the use of the GoLive Extend Script SDK with version 5.0 of Adobe GoLive.

Please be aware that both the SDK and this book are preliminary and subject to change. Visit <http://www.adobe.com/> for updated versions of this document and additional code samples.

ADOBE SYSTEMS INCORPORATED
Corporate Headquarters
345 Park Avenue
San Jose, CA 95110-2704
(408) 536-6000

Draft 1.0b7 • July 7, 2000

10

Adobe® GoLive™ 5.0 Extend Script Programmer's Guide for Windows® and Mac OS.

This manual, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. The content of this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in this documentation. Except as permitted by such license, no part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Acrobat, the Acrobat logo, and GoLive are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and other countries.

Macintosh and Mac are trademarks of Apple Computer, Inc., registered in the United States and other countries. Microsoft, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and other countries. All other trademarks are the property of their respective owners.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. government end users. The software and documentation are “commercial items,” as that term is defined at 48 C.F.R. §2.101, consisting of “commercial computer software” and “commercial computer software documentation,” as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §227.7202-1 through 227.7202-4, as applicable, the commercial computer software and commercial computer software documentation are being licensed to U.S. government end users (A) only as commercial items and (B) with only those rights as are granted to all other end users pursuant to the terms and conditions set forth in the Adobe standard commercial agreement for this software. Unpublished rights reserved under the copyright laws of the United States.

Version History

Date	Draft #	Comments
07 July 00	1.0b7	first public beta draft

Contents

Welcome	17
About The GoLive Extend Script SDK	17
Compatibility Information.	17
About This Book	18
Who Should Read This Book	18
How To Use This Book.	18
Document Conventions	20

PART I: Programmer's Guide

Chapter 1 Getting Started.	23
Contents of the GoLive Extend Script SDK	23
The ReadMe.html File	23
The Samples Folder	24
The GoLive SDK Documentation Folder	25
Installing the GoLive Extend Script SDK.	25
Installing Adobe GoLive	25
Installing the SDK Sample Code and Documentation.	25
Enabling the Extend Script Module	26
Installing the Sample Extensions	27
Configuring GoLive for Extend Script Development.	29
Enabling the JavaScript Shell palette	30
Examining the Sample Extensions.	31
Extensions to the Special Menu	31
Custom Menus.	31
Custom Palette Tabs and Custom Palette Items.	32
Executing JavaScript Code in the GoLive Environment	33
Setting the JavaScript Timeout	33
Accessing Page Elements From JavaScript	33
JavaScript Objects in the GoLive Object Model	34
Name Property and Name Attribute	35
JavaScript Object Arrays	35

Comparing Objects	36
Anatomy of an Extend Script Extension	37
Creating An Extend Script Extension Module	38
Creating the Extension Folder	38
Creating the Main.html File	38
Adding SDK Tags and JavaScript Functions to the Module	39

Chapter 2 Menus 41

Custom Menus	41
Overview	42
Adding the Menu Bar Tag	42
Defining the Menu	42
Defining Menu Items	43
Implementing the menuSignal Function	44
Assigning Keyboard Shortcuts To Menu Items	45
Multiple Custom Menus	47
Submenus	47
Setting A Menu Item's Checked State Explicitly	48
Setting a Menu Item's Enabled State Explicitly.	48
Setting The State of A Menu Item Automatically	49
Adding Items to the Special Menu	51

Chapter 3 Dialogs and Palettes 53

Modal Dialog Windows	53
Defining the Modal Dialog Window	54
Defining Dialog Content	55
Displaying the Dialog	59
Implementing the controlSignal Function	60
Floating Palettes	61
Using the Dialog Editor Extension	63

Chapter 4 Custom Elements 67

Overview	67
Tags For Creating Custom Elements	67
Custom Box Event-Handling Functions	68
Development Overview	68
Defining A Custom HTML Tag	69

Defining the Custom Tag's Palette Icon and HTML Content	70
Installing A Custom Entry In the Objects Palette	72
Adding Palette Entries to a Built-in Tab	73
Adding Palette Entries to a Custom Tab	74
Basic Custom Boxes	74
Initializing the Custom Box	75
Displaying the Custom Box	75
Inspecting the Custom Element	78
Resizing Custom Boxes	82
Built-In Undo Support	82
Drawing Custom Controls	83
Updating A Control's Appearance Immediately	83
Redefining Existing Tags	83

Chapter 5 Manipulating Document Objects 85

The Markup Tree	85
JavaScript Access to the Markup Tree	87
Selections	88
Retrieving the Current Selection	89
Setting the Current Selection	90
Manipulating Elements Programmatically	90
Supporting the Undo and Redo Commands	95
Creating the Undo Object	95
Initializing the Undo Object	96
Implementing the undoSignal Function	97
Accessing the Document History	98

Chapter 6 Files 99

Creating a File Object	99
Built-in Access to Commonly-Used Folders	99
Creating A File Object Explicitly	100
Testing For the Presence of a File or Folder	100
Determining What the File Object Represents	100
Creating A Folder Programmatically	101
Retrieving Files Programmatically	101
Retrieving A File's Location	101
Moving Files and Folders	101

Copying Files and Folders	102
Uploading Files To Remote Volumes	102

Chapter 7 Additional Topics 103

Working With Pictures	103
Creating Pictures.	103
Deleting Pictures.	103
Releasing Saved JavaScript References.	103
Timed Tasks	104
Persistent Data	104
Progress Bars	105
Localization.	106

Chapter 8 Debugging. 109

Integrated JavaScript Source Debugger	109
Enabling The Integrated Debugger and Other Debug Services	109
Script Debugger Window.	109
Setting Breakpoints	112
Script Breakpoints Window	113
Debugger Object (\$)	114
JavaScript Shell Palette	115

PART II: Reference

Chapter 9 Tags 119

Modules	119
jsxmodule	119
Locales	120
jsxlocale	120
Dialogs	120
jsxdialog	121
jsxpalette.	121
jsxcontrol.	122
Palette Items and Foreign Tags	124
jsxpalettegroup.	125
jsxpalettentry.	126

img	126
jsxelement	126
jsxinspector	128
Custom Element Example	128
Menus	130
jsxmenubar	131
jsxmenu	131
jsxitem	131

Chapter 10 Objects 133

Global Properties and Functions.	133
Global Properties	133
Global Functions.	134
write	135
GlobalPrefs Object.	138
Prefs Object	138
Application Object	138
Application Object Properties	138
Application Object Functions	140
Document Object	141
Document Object Properties	141
Document Object Functions	142
Module Object	143
Module Object Properties	143
Module Object Functions	143
Link Object	144
Link Object Properties	144
Link Object Functions	144
Box Object	145
Box Object Properties	145
Box Object Functions	146
Collection Object.	147
Collection Object Properties.	147
Collection Functions	147
Picture Object	147
Picture Object Properties	147
Picture Object Functions	147

Control Object	148
Control Object Properties	148
Control Object Functions.	149
Dialog Object.	150
Dialog Object Properties.	150
Dialog Object Functions	150
Draw Object	151
Draw Object Properties	151
Draw Object Functions.	151
Markup Object	152
Markup Object Properties	153
Markup Object Functions	153
Menu Object	154
Menu Object Properties	154
Menu Object Functions	154
MenuItem Object	155
MenuItem Object Properties.	155
MenuItem Object Functions	155
Selection Object	156
Selection Object Properties	156
Selection Object Functions	156
Undo Object	157
Global Undo Functions.	157
Undo Object Properties	157
Undo Object Functions.	158
History Object	158
History Object Properties	158
SiteReference Object.	159
SiteReference Object Properties.	159
SiteReference Object Functions	160
SiteCollection Object.	161
Site Collection Object Properties	161
Site Collection Object Functions.	161
File Object	162
File Object Properties	162
File Object Functions	163
\$ Object (Debugger Object)	165

\$ Object Properties	165
\$ Object Functions.	166

Chapter 11 Event-Handling Functions 167

Global Functions	167
initializeModule	167
terminateModule.	167
Custom Boxes	167
parseBox.	167
drawBox	168
boxResized	168
inspectBox	168
Controls	169
Custom Controls.	170

Chapter 12 C and C++ APIs For Use In External Binary Libraries 171

C API Synopsis	172
C++ API Synopsis	173
Data Types	173
JSValue pointer	173
JSValueType Scalar Types	173
JSANativeMethod Type	174
JSADrawInfo Struct	174
Initialization and Termination Functions	175
JSA_INIT Macro	175
JSAMain	175
JSARegisterFunction	176
JSAExit	176
Accessor Functions	177
JSGetValueType	177
JSValueToInt	178
JSValueToBool	178
JSValueToString	179
JSValueToDouble	179
JSIntToValue	179

JSABoolToValue	180
JSAStrngToValue	180
JSADoubleToValue.	180
JSASUndefinedToValue	181
JSASetError	181
JSASEval	181

PART III: Appendixes

Appendix A Using External Libraries 185

Benefits of External Libraries.	185
External JavaScript Libraries.	186
Implementing A JavaScript Library	186
Installing An External JavaScript Library.	186
Calling JavaScript Library Functions.	187
Implementing External Binary Libraries	187
Including C Libraries	187
Initializing the JavaScript Engine.	188
Defining External Library Functions	189
Registering External Functions	192
Implementing Optional Termination Code	192
Building An External Binary Library	193
Installing An External Binary	193
Calling C and C++ Library Functions From JavaScript.	193
Measuring Performance	194

Appendix B Sort Order Tables 195

Window Menu Items	195
Objects Palette Entries.	196

Glossary 199

Index 201

List of Figures

Figure 1.1	Enabling the Extend Script module	26
Figure 1.2	Extend Scripts folder with SDK installed	28
Figure 1.3	GoLive menu bar with SDK installed	28
Figure 1.4	JavaScript Shell palette	30
Figure 1.5	Markup Tree and Selections submenus of the Special menu	31
Figure 1.6	SDK Test menu	31
Figure 1.7	Server Side Includes palette tab and palette items	32
Figure 1.8	Dialog Editor palette tab and palette items	32
Figure 1.9	Contents of the Custom Box sample extension	37
Figure 2.1	The Hello, GoLive! menu	42
Figure 2.2	Alerts displayed by the Hello example	44
Figure 2.3	Submenu with its own submenu	47
Figure 2.4	Enabled and disabled menu items	49
Figure 2.5	Custom items appended to the Special menu	52
Figure 3.1	Positioning controls in dialog's coordinate plane	56
Figure 3.2	Type attribute specifies appearance and behavior of jsxcontrol object	57
Figure 3.3	Modeless dialog, floating window, or palette	61
Figure 3.4	Sort order in Window menu	63
Figure 3.5	Choosing the Dialog Editor palette	64
Figure 3.6	Selecting layout grid activates its inspector window	64
Figure 3.7	Layout view of typical dialog content	65
Figure 3.8	Source Code window with dialog code highlighted	66
Figure 4.1	Display attribute of jsxpalettentry	70
Figure 4.2	Custom tab in Objects palette.	72
Figure 4.3	Inspecting the Attributes of a Custom Element	78
Figure 5.1	HTML Outline view	86
Figure 5.2	Markup Tree window shows objects that contain current selection	86
Figure 7.1	Progress Bar	105
Figure 7.2	Busy Bar	105
Figure 8.1	Script Debugger window	110
Figure 8.2	Script Breakpoints window	113
Figure 8.3	JavaScript Shell Palette	115



Draft



List of Tables

Table 2.1	Translation of Modifier Keys to Alternate Platforms	46
Table 5.1	Identifiers for Document and markup objects	87
Table 7.1	Translation table example	106
Table B.1	Codes used to sort Window menu items	195
Table B.2	Codes used to sort Objects palette entries	196

Draft



Draft



List of Examples

Draft



Draft



Welcome

This Preface describes the use of this book, and provides information on Adobe programs for GoLive extension developers.

About The GoLive Extend Script SDK

The GoLive Extend Script SDK enables you to extend the behavior and user interface of version 5.0 of Adobe GoLive. Using specialized tags provided by the SDK, and your own JavaScript code, you can use GoLive scripting and layout tools to create customized page elements and user interface items in the GoLive website design environment, including fully-featured

- Floating palettes (modeless windows) that provide drag-and-drop tools and page elements
- Modal dialogs that include text, graphics and controls.
- Menus and menu items.
- Custom HTML elements.

Optionally, Extend Script extensions can call custom libraries written in the C, C++ or JavaScript programming language; however, knowledge of C is not required to use this SDK.

Compatibility Information

This SDK requires the use of version 5.0 of Adobe GoLive. This version of GoLive supports version 1.4 of the JavaScript language. By default, GoLive uses the Netscape flavor of JavaScript 1.4; however, you can specify that it use the Microsoft® JScript flavor or the ECMA-262 flavor. For details, see the description of the `flavor` property in “[\\$ Object Properties](#)” on page 165.

Although the JavaScript language supports Unicode, the GoLive Javascript interpreter does not. GoLive implements JavaScript strings as 8-bit ASCII.

If you plan to create an external C or C++ library that your **Extend Script** extension can call, you'll need Microsoft Visual C++ 6.0 to create a dynamically-linked library (DLL) for use on Windows® platforms, and Metrowerks' CodeWarrior 5 Pro to create shared libraries for use on Mac OS platforms.

About This Book

This book describes how to add functionality and custom user interface elements to Adobe GoLive 5.0 using the GoLive Extend Script SDK.

Who Should Read This Book

This book is for anyone who wants to extend the capabilities of Adobe GoLive using JavaScript and the special markup tags that the GoLive Extend Script SDK provides. This book assumes that

- You know how to create pages and web sites in Adobe GoLive, as described in the *Adobe GoLive User Guide* for version 5.0 of Adobe GoLive.
- You have a working understanding of the HTML and JavaScript languages, and have written some of your own JavaScript scripts.

You don't need knowledge of the C or C++ programming languages unless you plan to write an external code library in one of these languages. The vast majority of GoLive extensions do not use such libraries. For more information, see [Appendix A, "Using External Libraries."](#)

How To Use This Book

The first part of this book is a programmer's guide. It introduces the special tags and JavaScript functions that the SDK provides, and then describes how to use them to create GoLive extensions. The second part of this book provides detailed reference information about the tags and the JavaScript objects that the SDK provides.

All extension developers should read [Chapter 1, "Getting Started"](#) and ["Custom Menus" on pages 41 - 47](#). Once you've assimilated this material, you can skip to the section of this book that describes the programming task at hand; for example, if you need to display a window in GoLive, read the required sections and then go to [Chapter 3, "Dialogs and Palettes."](#)

Chapter Summaries

Here's a more detailed summary of the information that each chapter in this book provides:

- ["Welcome"](#) is this Preface. It describes the content of this book and offers suggestions for its most effective use.
- [Chapter 1, "Getting Started,"](#) provides conceptual information you'll use to write JavaScript modules that extend the Adobe GoLive design environment. It also provides an SDK installation guide and then provides step-by-step instructions that guide you through the creation of the basic file structure you'll use to define any extension.
- [Chapter 2, "Menus,"](#) begins with a brief tutorial that describes how to add a custom menu to GoLive. Subsequent sections describe submenus, multiple menus and setting the state of menu items.
- [Chapter 3, "Dialogs and Palettes,"](#) describes how to create modal dialogs and floating palettes.

- [Chapter 4, “Custom Elements,”](#) describes how to define a custom element as an icon the user can drag from the Objects palette to a GoLive document.
- [Chapter 5, “Manipulating Document Objects,”](#) describes how to manipulate GoLive document objects or their corresponding source tags from JavaScript.
- [Chapter 6, “Files,”](#) describes how to manipulate local files and their data programmatically.
- [Chapter 7, “Additional Topics,”](#) describes additional features the SDK provides.
- [Chapter 8, “Debugging,”](#) describes the use of the integrated JavaScript debugger that the GoLive Extend Script SDK provides.
- [Chapter 9, “Tags,”](#) describes custom tags that the GoLive Extend Script SDK supplies. You use these tags to define dialogs, palettes, controls, and custom elements your extension adds to the GoLive environment.
- [Chapter 10, “Objects,”](#) describes the JavaScript objects, properties, and functions that the SDK provides.
- [Chapter 11, “Event-Handling Functions,”](#) describes optional functions and methods your extension can implement to respond to events such as those generated by the user’s interaction with your extension’s controls.
- [Chapter 12, “C and C++ APIs For Use In External Binary Libraries,”](#) describes data types and utility functions that optional external C and C++ libraries can use to exchange data with Extend Script extensions.
- [Appendix A, “Using External Libraries,”](#) describes how to write C functions your extension can call from JavaScript.
- [Appendix B, “Sort Order Tables,”](#) provides the numeric values GoLive uses to order menu items.

To facilitate access to its contents, this book also provides

- a table of [Contents](#)
- an [Index](#)
- an Acrobat® Catalog index file (`index.pdx`) that enhances searches of this PDF document.

NOTE: The index file and its associated folder are in the **GoLive SDK Documentation** folder with the PDF file you are reading now. You must enable the index file before you can use it; for more information, choose the **Help>Acrobat Guide** menu item in Adobe Acrobat, then click the **Searching Catalog Indexes** bookmark.

Document Conventions

This section describes typographical and naming conventions used in this document.

Typographical conventions

This book uses the typographical conventions described here.

Boldface font identifies the first use and definition of a term.

Courier font identifies code, such as JavaScript code, HTML code, filenames, and pathnames. In pathnames, the forward slash (/) is used as a directory separator; for example the `Samples/Main.html` notation refers to the `Main.html` file in the `Samples` folder.

Italic text identifies replaceable text in code; for example, the *myName* text in `name="myName"` represents a value you are expected to supply. Thus `name="myName"` represents code such as `name="Fred"`, or `name="Homer"`, and so on.

Blue underlined text signifies a hyperlink you can click to display related information.

GoLive menus and menu items are listed in **sans-serif bold font**. The **>** symbol is used as shorthand notation for navigating to menu items; for example the **Edit>Cut** item refers to the **Cut** item in the **Edit** menu.

Naming Conventions

This book uses the naming conventions described here.

- The names of all tags and objects that the SDK provides begin with the `JSX` prefix; for example, the SDK provides a `<jsxmenu>` tag you can use to define menus. To improve readability, this book omits the `JSX` prefix from class and object names; for example, the class this book calls `Module` is actually implemented as the `JSXModule` class.

Terminology

This section introduces terminology conventions used in this book.

Throughout this book, you'll see many references to tags and elements. It's important to differentiate between the two, because the GoLive Extend Script SDK allows you to define and manipulate both. In short, tags define elements; specifically,

- A **tag** consists of alphanumeric tokens enclosed by angle brackets (`<>`), as in the ``, ``, or `<H1>` tags. Tags that are used singly, such as the `` tag, are **unary tags**. Tags that must be used in pairs are **binary tags**; for example, the `<H1>` opening tag must always be paired with an `</H1>` closing tag. When this book refers to a binary tag, it names the opening tag only and assumes you understand that the presence of the closing tag is implied.
- A tag defines an **element** when you supply all of the attribute values required to define an instance of the entity the tag represents. Thus, an HTML document could supply multiple `` tags that define multiple unique IMG elements. Some tags don't require any attributes at all; for example, the `
` break element is complete just as it appears here.

Part I

Programmer's Guide

Draft



Draft

1

Getting Started

The GoLive Extend Script SDK enables you to add custom page elements, tools, and user interface items to the GoLive 5.0 environment. This chapter provides conceptual and practical information you'll need to start creating your own extensions to the GoLive environment using JavaScript, special tags provided by this SDK and, optionally, standard HTML tags.

The first part of this chapter introduces the SDK itself:

- [Contents of the GoLive Extend Script SDK](#)
- [Installing Adobe GoLive](#)
- [Installing the GoLive Extend Script SDK](#)
- [Examining the Sample Extensions](#)

The second part of this chapter describes the JavaScript environment that GoLive provides to **Extend Script** extensions:

- [Executing JavaScript Code in the GoLive Environment](#)
- [Accessing Page Elements From JavaScript](#)

The chapter concludes by describing the file and folder structure GoLive requires extensions to have:

- [Anatomy of an Extend Script Extension](#)
- [Creating An Extend Script Extension Module](#)

Contents of the GoLive Extend Script SDK

The **Adobe GoLive SDK 5.0r1** folder holds a `Readme.html` file, a `Samples` folder and a **GoLive SDK Documentation** folder.

The ReadMe.html File

Before installing the SDK, see the `ReadMe.html` file for late-breaking information concerning the particular version of the SDK you are about to install.

The Samples Folder

The **Samples** folder offers a taste of the sorts of customizations to GoLive that this SDK enables you to create:

- **Custom Box**
Creates a custom Objects palette icon that
 - defines a custom HTML tag.
 - adds a custom HTML element to the page when the user drops the icon on a GoLive document window.Also provides a custom Inspector window that enables the user to modify the properties of the custom element interactively.
- **Menus and Dialogs**
Creates custom menus, menu items, dialogs, and dialog controls.
- **Palettes**
Creates a floating palette window and palette menu. Also demonstrates how to call an external library by calling a function that the Binary API sample implements
- **Markup Tree**
Manipulates the JavaScript objects that represent page elements in the GoLive environment.
- **Dialog Editor**
A more complex example that provides several custom elements as **Objects** palette entries. You can drag these palette entries to a GoLive document window to design a custom dialog window your extension can use.
- **KeyMap**
Displays a character map that the user can click to insert characters in a GoLive document. Uses external images to customize its display according to whether it is run on a Mac OS or Windows host platform.
- **Binary API**
Files for creating optional compiled external libraries that the JavaScript code in an Extend Script extension can call. This folder contains
 - Interface (header) and implementation files for developing external libraries in the C or C++ languages
 - A Metrowerks CodeWarrior project for Mac OS platforms
 - A Microsoft Developer Studio 6.0 project for Windows platformsMost **Extend Script** extensions don't require external code libraries—you'll note that the quite full-featured sample extensions don't—but the GoLive Extend Script SDK does support their use. If your extension has specialized needs that require calling an external C library from JavaScript, you can use the files in the **Binary API** folder to create platform-specific external code libraries for Mac OS and Windows platforms in the C programming language.
- **SSI**
Objects palette items that add server-side includes to a GoLive document.

- **Common**
Repository for optional external libraries, such as your own library of JavaScript functions, or the **JSASample** library that the SDK supplies.
- **Component**
Menu items that convert one or more GoLive Components into separate editable items. This extension is pre-installed in the `Modules\Extend Scripts` folder. The **Detach component** menu item is enabled when you select a component in the current document. The **Detach all components** menu item is enabled when the current document contains one or more components.
For information on creating and working with GoLive Components, see the *Adobe GoLive 5.0 User's Guide*.

The GoLive SDK Documentation Folder

The **GoLive SDK Documentation** folder contains this PDF document and an Acrobat Catalog index to speed your searches for information. You must enable the index file before you can use it; for more information, choose the **Help>Acrobat Guide** menu item in Adobe Acrobat, then click the **Searching Catalog Indexes** bookmark.

Installing the GoLive Extend Script SDK

This section describes how to install the GoLive Extend Script SDK.

Installing Adobe GoLive

The GoLive Extend Script SDK requires version 5.0 of Adobe GoLive. If you haven't yet installed version 5.0 of Adobe GoLive on the computer you're going to use to create extensions, you should do so now. For instructions, see the *Readme* file that accompanies the Adobe GoLive 5.0 product distribution.

Installing the SDK Sample Code and Documentation

The **Adobe GoLive SDK 5.0r1** folder provides this documentation, the code samples it references, and additional code samples this book does not describe. This section describes how to install the core set of sample code and documentation.

Installing on Mac OS Platforms

To install the sample code and documentation on your computer, drag the entire **Adobe GoLive SDK 5.0r1** folder to your hard disk. Then go on to the sections [“Enabling the Extend Script Module” on page 26](#) and [“Installing the Sample Extensions” on page 27](#).

Installing on Windows Platforms

To install the sample code and documentation on your computer,

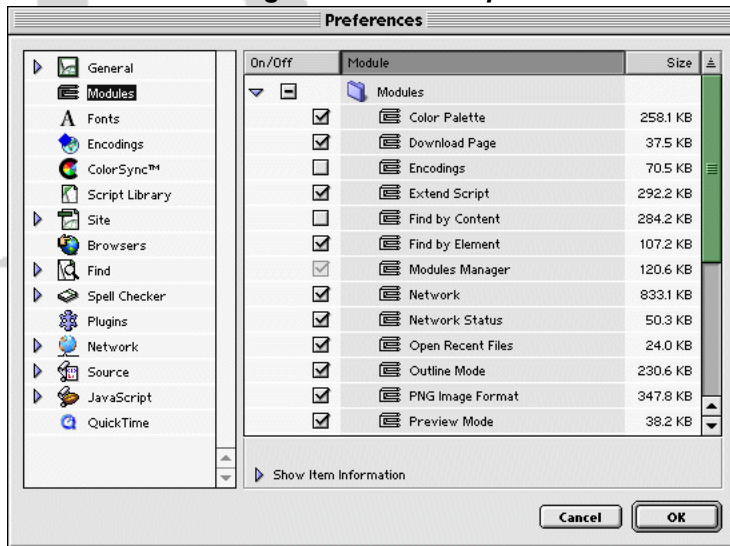
1. Run the Adobe GoLive product installer.
2. Select the **Custom** install option.
3. Click the **Adobe GoLive SDK** checkbox, placing a checkmark next to it.
4. Click the **OK** button.

Now, go on to the sections [“Enabling the Extend Script Module” on page 26](#) and [“Installing the Sample Extensions” on page 27](#).


Enabling the Extend Script Module

The GoLive Extend Script SDK requires use of the **Extend Script** module. If this module is not activated, you cannot load or run **Extend Script** extensions.

FIGURE 1.1 Enabling the Extend Script module



The GoLive installer activates this module when it installs GoLive; if this setting has been changed, you can take the following steps to enable the **Extend Script** module. You only need



to perform these steps once—the **Extend Script** module remains enabled in subsequent GoLive user sessions:

1. Open GoLive if it is not already running.
2. Open the **Edit>Preferences...>Modules** panel.
3. To enable the Extend Script module, click the **Extend Script** checkbox, placing a checkmark in it, as [Figure 1.1](#) illustrates.
4. Click OK to confirm your changes and dismiss the **Preferences** panel.
5. Quit GoLive and restart it.

You can take similar steps to enable other modules as required by the extension or site you are developing. To reduce startup time during extension development, you can disable unused modules as described in [“Configuring GoLive for Extend Script Development” on page 29](#).

Installing the Sample Extensions

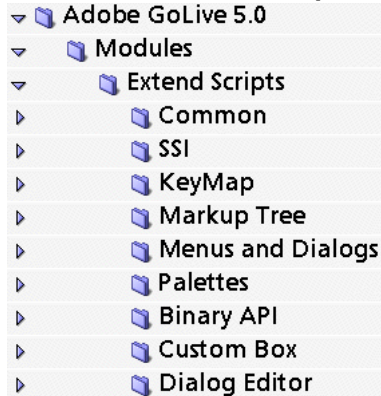
This section describes how to install an **Extend Script** extension in GoLive. You can use the procedure this section describes to install the sample extensions that the SDK provides, as well as your own extensions, or those provided by third parties.

Each folder in the `Samples` folder holds a different example of an **Extend Scripts** extension. It is recommended that you install the following core set of sample extensions, as this programmer’s guide refers to them frequently:

- **Custom Box**
- **Menus and Dialogs**
- **Palettes**
- **Markup Tree**
- **Dialog Editor**
- **KeyMap**
- **Binary API**
- **Common**

Once you’ve become familiar with the use of the tags, scripts and objects these samples illustrate, you can remove any or all of them, as you prefer.

FIGURE 1.2 *Extend Scripts folder with SDK installed*



Installing an Extension

To make an extension available to GoLive,

1. Quit GoLive if it is running.
2. Drag the extension's folder from the Samples folder to the Adobe GoLive 5.0/Modules/Extend Scripts folder.

To install all of the sample extensions, drag the entire contents of the Samples folder to the Extend Scripts folder.

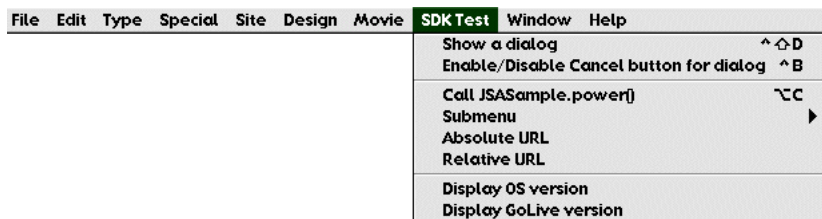
If you prefer not to install all of the sample extensions that the SDK provides, you can install just the extensions shown in [Figure 1.2 on page 28](#). Most of the code examples in this book are based on this core set of extensions.

3. Start GoLive.

When GoLive starts up, it loads all of the extensions present in the subfolders of the **Extend Scripts** folder. GoLive can load as many extensions as available RAM permits.

If you installed the core set of sample extensions that [Figure 1.2](#) depicts, the GoLive menu bar should now include the **SDK Test** menu shown in [Figure 1.6](#).

FIGURE 1.3 *GoLive menu bar with SDK installed*



If the **SDK Test** menu is not present, try the following possible solutions:

- Make sure the **Extend Script** module is activated as described in [“Enabling the Extend Script Module” on page 26](#).
- Make sure the **Menus and Dialogs** example folder is in the Adobe GoLive 5.0/Modules/Extend Scripts/ folder.
- Make sure the **Menus and Dialogs** folder holds a `Main.html` file.

NOTE: If these suggestions don’t resolve the problem, reinstall GoLive and the SDK.

Uninstalling an Extension

Take the following steps to remove an extension:

1. Drag the extension’s folder out of the **Extend Scripts** folder
2. Restart GoLive.

Configuring GoLive for Extend Script Development

Developing GoLive extensions is an iterative process that requires you to restart GoLive whenever you need to load a new version of the extension you’re developing. In order to reduce the time required for GoLive to start up, you can disable modules that your extension does not use.

To disable a module,

1. Uncheck it in the **Edit>Preferences...>Modules** panel.
2. Restart GoLive.

The precise set of modules you can disable successfully depends on the features your extension or site uses—you cannot disable a module your site or your extension requires for its functionality. Here’s a suggested list of modules you might consider disabling to reduce startup time during extension development:

Download Page	Dynamic Link	Network [*]	Network Status
PNG Image Format	QuickTime Module	Site Design	Smart Links
Smart Objects	SWF Module	WebDAV	WebObjects
Encodings [†]			

^{*}Do not disable the **Network** module if any extension calls the `get` or `put` methods of the `JSXFile` object.

[†]Do not disable the **Encodings** module in Japanese versions of GoLive.

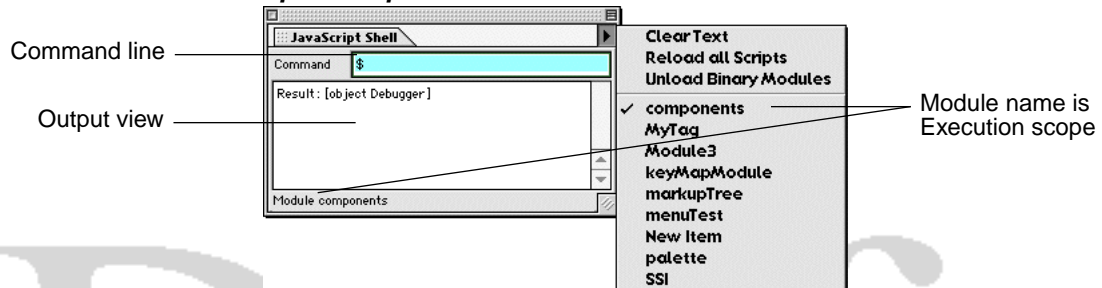
NOTE: Do not disable the **Extend Script** module; if this module is not enabled, GoLive cannot load or run **Extend Script** extensions.

For descriptions of these modules, see the “Adobe GoLive Module Reference” section of the *Adobe GoLive 5.0 User’s Guide*.

Enabling the JavaScript Shell palette

One of the best ways to become familiar with the JavaScript environment in GoLive is to experiment by entering JavaScript expressions into the **JavaScript Shell** palette. This window provides a command line you can use to communicate interactively with the JavaScript engine built into GoLive. If you've installed all of the samples provided by the SDK, the **Window>JavaScript Shell** menu item should display a palette similar to the one [Figure 1.4](#) depicts.

FIGURE 1.4 JavaScript Shell palette



For a more detailed description of this window, see [“JavaScript Shell Palette” on page 115](#). Later on in this chapter, [“Executing JavaScript Code in the GoLive Environment” on page 33](#) describes the GoLive JavaScript environment in detail.

JavaScript Shell Palette Problems

If you get any kind of response from GoLive in the output view of the JavaScript shell palette when you press the Enter key, you can skip this section. Both of the following problems are resolved by enabling debugging in at least one module in the **Extend Scripts** folder

- If no currently-installed extension modules enable debugging services, the **JavaScript Shell** item does not appear in the **Window** menu.
- If the **JavaScript Shell** palette was open when the last user session with GoLive ended, this window is opened again the next time GoLive starts. In this case the JavaScript shell window may be open but not responsive to input.

Look for the name of the current module in the lower-left corner. If no text appears there, no modules currently enable debugging services. Debugging services are enabled when at least one Extend Script module contains the debugger statement or the debug attribute to `<jsxmodule>` tag.

NOTE: To enable debugging services quickly and easily, drag any of the sample extensions supplied by the SDK into your **GoLive>Modules>Extend Scripts** folder and restart GoLive.

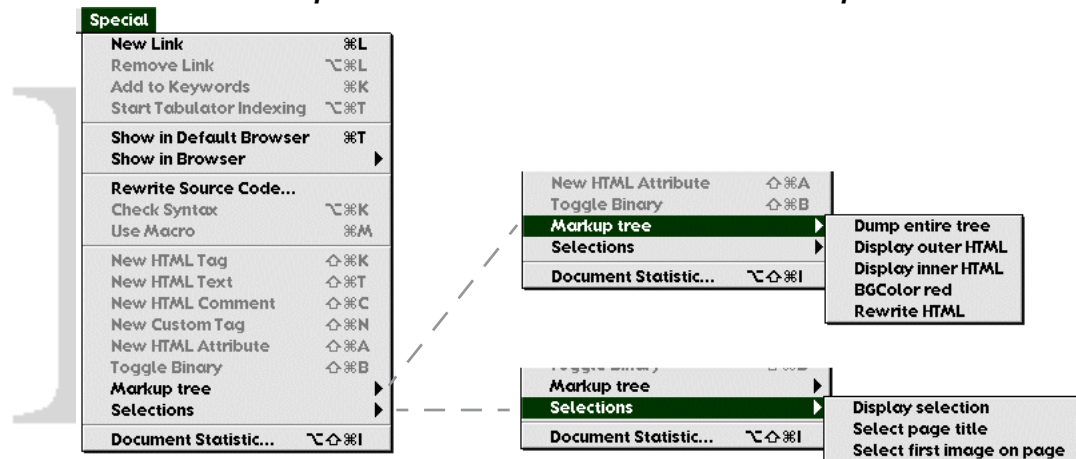
Examining the Sample Extensions

The SDK provides examples of extensions that install a custom menu, custom menu items, custom palette tabs, and custom palette objects. This section provides a brief overview of the user features that the SDK sample extensions install in the GoLive environment.

Extensions to the Special Menu

The **Markup Tree** example installs custom **Markup Tree** and **Selections** items in the **Special** menu, as [Figure 1.5](#) illustrates. These items are submenus that holds additional menu items, as shown in Figure 1.5.

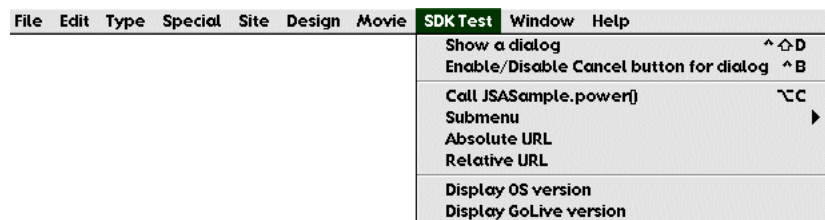
FIGURE 1.5 Markup Tree and Selections submenus of the Special menu



Custom Menus

The **SDK Test** menu is provided by the **Menus and Dialogs** sample extension. If you haven't done so already, take a moment now to experiment with the **SDK Test** menu and its menu items.

FIGURE 1.6 SDK Test menu

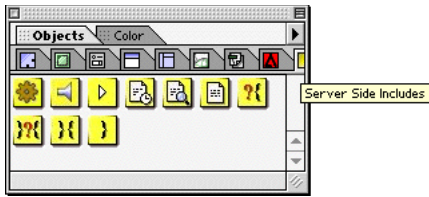


Custom Palette Tabs and Custom Palette Items

The core set of sample extensions installs three new tabs in the Objects palette:

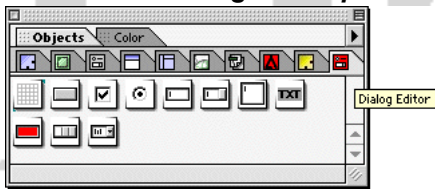
- The **Custom Box** example installs the **Sample Tag** tab and a palette item you can drag onto the page to create an instance of a custom page element.
- The **SSI** example installs the **Server Side Includes** tab and several palette items you can drag onto the page to add server-side includes to your site. [Figure 1.7](#) depicts the Server Side Includes tab and its associated palette items.

FIGURE 1.7 *Server Side Includes palette tab and palette items*



- The **Dialog Editor** example installs the Dialog Editor tab and several palette items you can drag onto the page to create common dialog box controls such as buttons, checkboxes and text fields. [Figure 1.8](#) depicts the Dialog Editor tab and its associated palette items.

FIGURE 1.8 *Dialog Editor palette tab and palette items*



NOTE: A dialog created by the Dialog Editor sample is not a standalone extension—it is code meant to be incorporated in an extension that displays the dialog as part of its user interface. For more information on displaying dialogs, see [Chapter 3, “Dialogs and Palettes.”](#)

Executing JavaScript Code in the GoLive Environment

GoLive provides two design features intended to minimize problems caused by misbehaving JavaScript code.

- If the SDK detects an error in a drawing function, such as in the `drawControl` or `drawBox` function, it does not call this function again until you close and reopen the document that defines the function. Without this feature, an error in a drawing function would cause GoLive to throw an endless stream of errors, because drawing functions are called whenever a JavaScript object must be redrawn.
- You can specify the amount of time GoLive waits for JavaScript code to return control before GoLive exits the current script unconditionally. Without this feature, an infinite loop or other failure in JavaScript code could halt GoLive indefinitely.

Setting the JavaScript Timeout

By default, GoLive waits forever for a JavaScript function call to complete. If you are not confident that a JavaScript function can complete its task in a reasonable amount of time, you may prefer to specify the amount of time GoLive waits for a response from it. Each extension can specify its own timeout that GoLive uses to execute that extension's scripts.

To set the script execution timeout, add to your extension's `Main.html` file a `jsxmodule` tag that provides a `timeout` attribute. The value of this attribute is the number of seconds GoLive waits for a script to return control before it exits the script. Values of 0 or `false` restore the default behavior of never timing out.

```
<html>
  <body>
    // give scripts ten seconds to complete before unconditional exit
    <jsxmodule timeout=10>
      // scripts and SDK tags go here
    </body>
  </html>
```

NOTE: An external library that calls an Extend Script extension can specify a separate temporary script execution timeout for each JavaScript call. For more information, see the description of the `JSAEval` function in [Appendix A, "Using External Libraries"](#).

Accessing Page Elements From JavaScript

The JavaScript interface to GoLive provides access to page elements in a way that JavaScript programmers will find familiar. However, the GoLive Extend Script SDK is also intended to be useful to the C programmer who doesn't want to dive too deeply into the fine print of JavaScript. Thus most page elements in the GoLive environment can be accessed in multiple ways. This section describes how GoLive generates the JavaScript representation of the markup tags in a document, and various ways to access these objects from JavaScript.

JavaScript Objects in the GoLive Object Model

When GoLive reads an HTML document, it creates an internal representation of the tags in the document and leaves the contents of the document unchanged. When GoLive interprets a markup tag, it creates two sets of objects:

- An internal representation of the markup element that the tag and a particular set of attribute values defines. This internal representation is not created in JavaScript and JavaScript callers cannot access it directly.
- A set of JavaScript objects that provide access to the internal representation of the markup element.

The JavaScript objects in this representation take the form of a binary tree of objects known as the **markup tree**. Individual objects in the tree are called **markup elements**. The Javascript expression `document.element` provides access to the Markup object that is the root of a document's markup tree.


At any time, changes to a document that holds a markup element can cause GoLive to discard its internal data structures and generate new ones. This operation is called **reparsing**. Any of the following situations can cause GoLive to reparse the document:

- Adding or deleting elements in a GoLive document window. The user or an extension can add or delete elements.
- Adding or deleting pages to a site may potentially cause GoLive to reparse all of the documents in a site.
- The programmatic actions of your extension or others can also change one or all documents in a site in ways that require GoLive to reparse.
- Even simple user interaction with the window in which a document appears—for example, resizing the document window in a way that changes the orientation of page elements—can cause GoLive to reparse the document.

In the GoLive environment, JavaScript objects don't hold any data or functionality themselves. They simply enable JavaScript access to data or functionality that is actually provided by an internal representation that GoLive maintains. Thus, it's extremely important that you manage JavaScript references to document objects correctly in order to avoid invalid object references that reparsing operations may cause:

- Variables that hold object references must be reinitialized to reference current objects whenever GoLive reparses the document holding the tags that define these objects. GoLive reparses a document
 - whenever the user changes it, or
 - whenever GoLive or an extension calls the document's `reparse` method.
- Avoid saving document references in global variables. Unless you maintain the reference scrupulously and exercise extreme care in its use, it can outlive the document it references.

IMPORTANT: *Variables holding JavaScript objects are invalid after the document is changed. Whenever the document changes, you must reinitialize such variables to reference current JavaScript objects.*



When GoLive reparses, it updates the JavaScript representation of the markup tree, but it may not be able to update your variables that hold JavaScript objects. As a result, these saved JavaScript objects may contain references to internal data structures that no longer exist. When a JavaScript object attempts to access a nonexistent internal data structure, GoLive returns an error.

Although the SDK authors have added numerous checks to avoid such crashes, performance considerations make it impractical to validate every pointer in a large markup tree. Thus, to preserve acceptable performance, you must take responsibility for using saved JavaScript objects correctly.

In summary,

- If you created a JavaScript object yourself by using the `new` operator on its constructor, you can save the object in a variable and use that variable as you would normally; however, you must set this variable to `null` before GoLive unloads your extension. For more information, see [“Releasing Saved JavaScript References” on page 103](#).
- You must reinitialize your own references to automatically-generated JavaScript objects whenever the document is reparsed. Attempting to use an invalid object reference causes GoLive to return an error.

Name Property and Name Attribute

JavaScript objects representing most GoLive page or site elements can be retrieved by name. A JavaScript object’s name corresponds to the value of the `name` attribute specified by the markup element GoLive interpreted to create it.

Usually, the `name` property of an element’s corresponding JavaScript representation is specified by the `name` attribute of the tag that caused GoLive to create the JavaScript object. If you omit the `name` attribute from the tag, GoLive usually provides a default `name` attribute for you; however, it is recommended that you choose your own names, if only for the convenience of knowing exactly which name is associated with a particular object.

IMPORTANT: *To ensure reliable name-based access to JavaScript objects, the `name` attribute of each tag your extension uses must be unique within the JavaScript namespace.*

JavaScript Object Arrays

In addition to the access afforded by each JavaScript object’s unique `name` property, the global namespace makes most objects available as the elements of various arrays it maintains in the global namespace. For example, the global namespace contains an object named `menus` which you can use to access all menus defined by all currently-active extensions.

Sub-elements are made available as the properties of a parent element; for example, a menu item does not appear in the global namespace—it is a property of the menu in which it appears. The parent menu may be accessed by means of its unique `name` attribute, or by means of the `menus` object that GoLive always creates in the global namespace.

Because array elements can be addressed by numeric index or by name, the following lines of JavaScript are equivalent.

```
menus ["sample"].items ["item1"]
menus ["sample"].items [0]
menus ["sample"].item1
sample.item1
```

NOTE: Most of the arrays that the SDK provides are Collection objects; even though these objects are arrays, their elements are not accessible by numeric index, only by name.

The `menus` array holds all of the menus and menu items added to GoLive by Extend Script extensions. The following line of JavaScript retrieves the `item1` menu item by name from the `sample` menu, which it also retrieves by name.

```
menus ["sample"].items ["item1"]
```

The `menus ["sample"]` expression gets the menu named `sample` by name from the global `menus` array. The `sample` menu's `items` property holds the array of menu items. To retrieve the `item1` element from this array by name, we use the same technique we used to get the `sample` element from the global `menus` array.

Alternatively, you can retrieve the `item1` menu item as a property of the `sample` menu, as in the next line of JavaScript.

```
menus ["sample"].item1
```

GoLive also makes each menu available as a JavaScript object in the global namespace; thus, the following simple line of JavaScript provides yet another way to retrieve the first item in the `sample` menu.

```
sample.item1
```

IMPORTANT: *To avoid unpredictable results, the name of each element your extension defines must be unique within the global namespace.*

Comparing Objects

To ascertain an object's identity, you can compare the value of its `name` property to a known value, such as a string or a reference to a global object. For example, you could test the name of a menu item in any of the following ways:

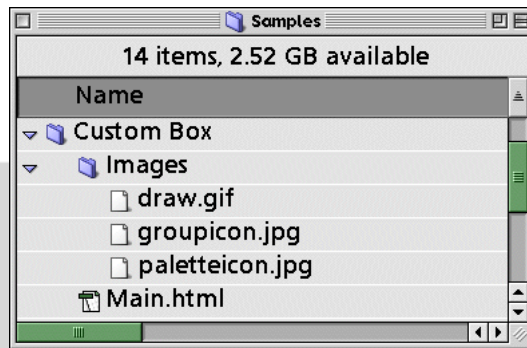
```
if (item.name == "item1")
if (item == menus ["sample"].items ["item1"])
if (item == sample.item1)
```

Anatomy of an Extend Script Extension

Each GoLive extension is defined by a `Main.html` file that resides in a subfolder of the **Extend Scripts** folder. For example, each of the subfolders the SDK installs in the **Extend Scripts** folder contains a `Main.html` file that defines an extension to the GoLive environment.

All external files the extension requires—images, plug-ins, and the like—must reside in the same folder as the extension's `Main.html` file; for example, in [Figure 1.9](#), the **Custom Box** extension's folder holds external `.gif` and `.jpg` image files in addition to the extension's `Main.html` file.

FIGURE 1.9 Contents of the Custom Box sample extension



Your extension's `Main.html` file holds any HTML and JavaScript necessary to define the extension. At startup time, GoLive interprets these tags and scripts to create an extension in the GoLive environment.

Most of the HTML in the `Main.html` file consists of special tags that the SDK provides. You use these tags to define the extension's menus, controls, inspectors, palettes and custom tags. You can combine the special SDK tags with standard HTML tags as necessary.

The JavaScript code in the `Main.html` file consists of your own functions and your implementations of GoLive [Event-Handling Functions](#). GoLive calls these functions at specific times to make your extension perform specific tasks. Your extension implements its own versions of these functions to provide the functionality GoLive requires at these times.

GoLive calls most of these functions in response to user events. For example, when the user interacts with an extension's custom menu, dialog, or palette, GoLive calls the appropriate event-handling function. If the extension provides that function, GoLive executes it; otherwise, the extension ignores the call to that function. For example, when the user selects one of your extension's menu items, GoLive sends the `menuSignal` message to your extension, along with some information describing the menu item chosen. If your extension provides a `menuSignal` function, GoLive executes it in response to the user's menu choice; if not, your extension simply ignores the `menuSignal` message.

Now that you have an idea of how tags, scripts, and event-handling functions work together in the GoLive environment, you're ready to get down to the business of creating extensions.

Creating An Extend Script Extension Module

This section describes how to create the file and folder structure GoLive expects Extend Script extensions to have.

Creating the Extension Folder

Take the following steps to begin creating an extension:

1. Create a new folder to hold the extension's files. The rest of this book refers to this folder as the **Extension Folder**.
2. Give the extension folder a unique name.
3. Place the extension folder in the Adobe GoLive/Modules/Extend Scripts/ folder.

Creating the Main.html File

Every Extend Script extension takes the form of a Main.html file that resides in its own uniquely-named folder in the **Extend Scripts** folder. You can take the following steps to create the Main.html file for a new extension,

1. Create a text file named Main.html.
2. Place the Main.html file in its own uniquely-named folder inside the **Extend Scripts** folder.
3. Add the following text to the file:

```
// Skeleton of the Main.html file
<html>
  <body>
    // Tags that define your menu & menu items will go here.
  </body>
</html>
```

Like any HTML file, its first tag must be the <HTML> tag, and its last tag the </HTML> tag.

The HTML that defines your extension will reside in the body of the Main.html file, so you'll need to add a <BODY> tag just after the <HTML> tag. Of course, you also need to add the closing </BODY> tag just before the closing </HTML> tag.

4. Save the file.

Now this file is ready to hold the tags and scripts that define your extension.

This set of instructions describes a minimal Main.html file that could be created in a simple text editor. If you use GoLive to create your Main.html file, you'll also see <META> tags and a default <TITLE> element in the HTML files it creates. To specify the file GoLive uses as the template for a new document, check the **New Document...** checkbox in the **Edit>Preferences>General** panel, then click the **Select...** button to specify the file.

Adding SDK Tags and JavaScript Functions to the Module

To create a GoLive extension, you'll add tags and scripts to the body of a `Main.html` file created as described in [“Creating An Extend Script Extension Module” on page 38](#).

For a tutorial that describes how to add a custom menu to the GoLive menu bar, see the first part of [“Custom Menus” on page 41](#). Even if you don't plan on adding a custom menu to GoLive, you should read this tutorial to gain an understanding of how to use tags and scripts to create a simple extension.

After you've read (or, ideally, worked through) this material, you should be able to use any other section of the book as needed to meet your extension development goals.

Draft



Draft

2

Menus

This chapter describes how you can use the GoLive Extend Script SDK to perform tasks such as

- Creating a custom menu
 - Displaying an alert
 - Put up a dialog that displays an external graphic or custom text, perhaps using custom fonts, colors, and so on.
- Adding custom menu items to the Special menu
- Adding submenus
- Assigning keyboard shortcuts to custom menu items
- Setting the enabled state of a menu item
- Setting the checked state of a menu item

Pop-up menus, which appear in dialog windows, are described in [“Defining Dialog Content” on page 55](#).

Custom Menus

This section describes the steps required to add a custom menu to the GoLive design environment:

- [Adding the Menu Bar Tag](#)
- [Defining the Menu](#)
- [Defining Menu Items](#)
- [Implementing the menuSignal Function](#)

These required steps are presented in tutorial fashion; once you’ve read (or, ideally, worked through) these sections, you should be able to skip to other parts of the book as needed to meet your extension development goals.

Subsequent sections in this chapter describe the following optional topics:

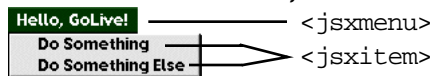
- [Multiple Custom Menus](#)
- [Submenus](#)
- [Setting A Menu Item’s Checked State Explicitly](#)
- [Setting a Menu Item’s Enabled State Explicitly](#)
- [Setting The State of A Menu Item Automatically](#)

Overview

You can define a custom menu like the one in [Figure 2.1](#) using only three tags and one JavaScript function:

- The [jsxmenubar](#) tag wraps the definition of all the HTML that adds menus to the GoLive menu bar.
- The [jsxmenu](#) tag defines a custom menu that appears to the left of the **Window** menu in the GoLive menu bar.
- The [jsxitem](#) tag defines a custom menu item that appears in a custom menu.
- The [menuSignal](#) function performs the menu item's task.

FIGURE 2.1 The Hello, GoLive! menu



Adding the Menu Bar Tag

The `<jsxmenubar>` opening tag must precede all of the tags that define custom menus and custom menu items. Similarly, the `</jsxmenubar>` closing tag must close the HTML that defines custom menus and custom menu items. After you've added these tags, your `Main.html` file should look like the following example.

```
// Main.html file for menu example
<html>
  <body>
    <!-- The Hello GoLive Menu ----->
      <jsxmenubar> // opens definition of all menus and menu items
        // Tags that define your menu & menu items will go here.
      </jsxmenubar> // closes definition of all menus and menu items
    </body>
  </html>
```

Defining the Menu

Inside the `<jsxmenubar></jsxmenubar>` tags, you'll add the [jsxmenu](#) tag that defines the name of your custom menu. This tag is a binary tag; that is, it consists of an opening tag and a closing tag. Its syntax looks like this:

```
<jsxmenu name="name" title="Menu text"></jsxmenu>
```

The `title` property specifies the menu's title in the GoLive menu bar, while the `name` property specifies the name used to access the menu in the JavaScript namespace. Thus, the following tag defines that portion of the menu that appears in the menu bar.

```
<jsxmenu name="Hello" title="Hello, GoLive!"></jsxmenu>
```

If you omit the `name` property or neglect to supply a value for it, GoLive uses the value of the `title` property as the default value of the `name` property.

With the top level of your menu defined, your Main.html file should now look like the following example.

```
// Main.html file for Hello example
<html>
  <body>
    <!-- The Hello GoLive Menu ----->
    <jsxmenubar> // opens definition of all menus and menu items
      <jsxmenu name="Hello" title="Hello, GoLive!"></jsxmenu>
      // Tags that define your menu items will go here.
    </jsxmenubar> // closes definition of all menus & menu items
  </body>
</html>
```

Now you're ready to add the tags that define this menu's menu items.

Defining Menu Items

The `<jsxmenu></jsxmenu>` tags enclose one or more `<jsxitem>` tags. Each `<jsxitem>` tag defines a menu item that is to appear in the custom menu. The syntax line for this tag looks like the following example.

```
<jsxitem name="name" title="Item text" key = "text" dynamic>
```

The name property specifies this object's identifier in the JavaScript namespace, while the title property specifies the text that appears as the menu item.

The optional key attribute is described later in this chapter, in [“Assigning Keyboard Shortcuts To Menu Items” on page 45](#). The optional dynamic attribute is described later in this chapter, in [“Setting The State of A Menu Item Automatically” on page 49](#).

The following example defines two custom menu items.

```
<jsxitem name="doThis" title="Do Something">
<jsxitem name="doThat" title="Do Something Else">
```

The menu items appear in the menu in the same order that their definitions appear in the Main.html file. After adding the menu items, the Main.html file should resemble the following example.

```
// Main.html file for Hello example
<html>
  <body>
    <!-- The Hello GoLive Menu ----->
    <jsxmenubar> // opens definition of all menus
      <jsxmenu name="Hello" title="Hello, GoLive!"> // Hello menu
        <jsxitem name="doThis" title="Do Something"> // menu item
        <jsxitem name="doThat" title="Do Something Else" > menu item
      </jsxmenu> // closes definition of Hello menu
    </jsxmenubar> // closes definition of all menus
  </body>
</html>
```

To admire the fruits of your labor thus far, save and close the `Main.html` file. The next time you start GoLive, you should see the menu that [Figure 2.1](#) depicts; however, its menu items won't do anything until you write a `menuSignal` function that GoLive can call when the user chooses one of these menu items.

Implementing the menuSignal Function

When the user chooses a menu item, GoLive calls that menu's `menuSignal` function. This section describes how to write a `menuSignal` function that takes action in response to the user's choice of a menu item.

```
function menuSignal(menuItem)
{
    // code that acts on the user's choice of menu item
}
```

To determine which menu item the user chose, your `menuSignal` function can test the `name` property of the `jsxitem` object that GoLive passes as the argument to this function. The dot operator (`.`) provides access to the properties of a `jsxitem` object, just as it does for other JavaScript objects. Thus, to retrieve the name of the menu item the user chose, your `menuSignal` function can use the `menuItem.name` expression. The example `menuSignal` function immediately following uses the `menuItem.name` expression as the case expression of a switch statement that varies its actions according to the menu item the user chose.

```
function menuSignal(menuItem)
{
    switch (menuItem.name)
    {
        case "doThis":alert("You chose the Do Something item.");
                        break;
        case "doThat":alert("You chose the Do Something Else item.");
                        break;
        default:alert ("Something went wrong...");
    }
}
```

Because displaying an alert is one of the easiest things you can do with GoLive from JavaScript, each case in this example `menuSignal` function displays an alert that simply names the menu item chosen, as shown in [Figure 2.2](#).

FIGURE 2.2 Alerts displayed by the Hello example



The `menuSignal` function must reside within a `<SCRIPT></SCRIPT>` tag.

IMPORTANT: *Each `Main.html` file can contain only one set of `<SCRIPT></SCRIPT>` tags. All of your extension's scripts must reside within these tags.*

Your completed Main.html file that defines the **Hello, GoLive!** menu should resemble the following example.

```
<html>
  <body>
    <script>

      function menuSignal(menuItem)
      {
        switch (menuItem.name)
        {
          case "doThis":alert("You chose the Do Something item.");
                        break;
          case "doThat":alert("You chose the Do Something Else item.");
                        break;
          default:alert ("Something went wrong...");
        }
      }

    </script>

    <!-- The Simplest Menu ----->

    <jsxmenubar>
      <jsxmenu title="Hello, GoLive!">
        <jsxitem name="doThis" title="Do Something">
        <jsxitem name="doThat" title="Do Something Else">
      </jsxmenu>
    </jsxmenubar>
  </body>
</html>
```

If you've read or worked through this entire [Custom Menus](#) section to this point,

- You have now seen all of the tags and functions required to create a custom menu
- You've seen how to use SDK-provided tags and functions to create a simple extension.
- You can skip to any section in the remainder of this book that suits your extension development goals.

The remainder of this chapter describes optional topics related to custom menus.

Assigning Keyboard Shortcuts To Menu Items

You can use the optional `key` attribute to the [jsxitem](#) tag to assign keyboard shortcuts to your menu items. To specify modifier keys, use the identifiers `Ctrl`, `Shift`, `Alt`, `Opt`, or `Cmd` as required. For example, the first line in the following example assigns the `Ctrl-Shift-D` keystroke

to the **Do Something** menu item, while the second line assigns the Alt-E keystroke to the **Do Something Else** menu item.

```
<jsxitem name="doThis" title="Do Something" key = "Ctrl+Shift+D">
<jsxitem name="doThat" title="Do Something Else" key = "Alt+E">
```

GoLive remaps platform-specific modifier keys as [Table 2.1](#) specifies. For example, GoLive treats the Windows platform's Alt keystroke as the Option keystroke on Mac OS platforms. Similarly, GoLive treats the Mac OS Cmd keystroke as the Ctrl keystroke on Windows platforms.

TABLE 2.1 Translation of Modifier Keys to Alternate Platforms

Modifier Key (Native Platform)	Mac OS platforms	Windows platforms
Control/Ctrl (Windows)	Cmd	Ctrl
Command (Mac OS)	Cmd	Ctrl
Alt (Windows)	Opt	Alt
Option (Mac OS)	Opt	Alt
Shift	Shift	Shift

Using the ampersand (&) character in menu titles forces Windows platforms to underline the character that follows it, indicating to the user that the character is used as a hot key. On the Macintosh, these ampersand characters are removed from the string so they do not display. To display the ampersand itself on Windows, use two consecutive ampersands. When reading the `title` property, the ampersand characters are always removed from the original string to preserve compatibility across operating systems. For example, if you assign the string "&New" to the `title` property of a menu item it displays as **New** on Windows platforms and displays as New on Mac OS platforms. The value of the `title` property is "New" as retrieved from the menu item object.

If you assign a keyboard shortcut that is already in use by another menu item, results are unpredictable. You can take either of the following approaches to work around the problem:

- Change the conflicting shortcut in your extension's source code
 - Assign a new value to the conflicting key attribute in your extension's `Main.html` file.
 - Restart GoLive with the new `Main.html` file in the Extend Scripts folder
- Change the conflicting shortcut in the **Edit>Keyboard Shortcuts** dialog.

IMPORTANT: *The Keyboard Shortcuts dialog records all shortcuts it finds on disk. When you use this dialog to change the order of menu entries, existing shortcuts may be corrupted. To restore the original mappings of keystroke shortcuts, you must erase the Adobe Golive 5.0 Preferences file.*

On MacOS systems, the preferences file is located at
System Folder: Preferences: Adobe GoLive 5.0 Prefs

On Windows 98 platforms, the preferences file is located at
C:\WINDOWS\Application Data\Adobe\Adobe Golive\PrefFile.prf

On Windows NT platforms, the preferences file is located at

C:\WINNT\Profiles\UserName\AppData\Local\Adobe\Adobe GoLive\PrefFile.prf

Multiple Custom Menus

GoLive adds custom menus to the menu bar at the left of the Window menu. You cannot specify the order in which GoLive loads multiple custom menus; however, the items in a menu always appear in the order the Main.html file defines them.

Submenus

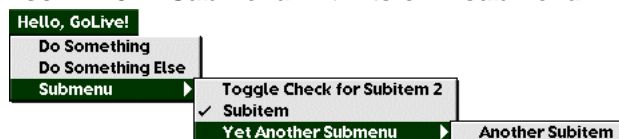
You can also use the [jsxmenu](#) element to define a submenu—just place the submenu's `<jsxmenu>` element amongst the `<jsxitem>` tags that define the menu's items, as the following example does

```
<!-- Submenu Example ----->
```

```
<jsxmenubar>
  <jsxmenu name="HelloMenu" title="Hello, GoLive!">
    <jsxitem name="doThis" title="Do Something" dynamic>
    <jsxitem name="doThat" title="Do Something Else">
    <jsxmenu name="subMenu" title="Submenu">
      <jsxitem name="toggle" title="Toggle Check for Subitem 2">
      <jsxitem name="sub2" title="Subitem">
      <jsxmenu name="anotherSubMenu" title="Yet Another Submenu">
        <jsxitem name="anotherSubItem" title="Another Subitem">
      </jsxmenu>
    </jsxmenu>
  </jsxmenu>
</jsxmenubar>
```

The submenu can include menu items that are themselves submenus. This code example produces the menu hierarchy that [Figure 2.3](#) depicts.

FIGURE 2.3 Submenu with its own submenu



Setting A Menu Item's Checked State Explicitly

You can set a `jsxitem` object's `checked` property to specify whether GoLive is to place a check mark next to the object's corresponding menu item. A value of `true` specifies that the menu item has a check mark, as the **Subitem** menu item in [Figure 2.3](#) does. In the following example `menuSignal` function, the `toggle` case sets this property to its opposite value each time the user chooses the **Toggle Check for Subitem 2** menu item.

```
<!-- Check mark Example ----->

<jsxmenubar>
  <jsxmenu title="Hello, GoLive!">
    <jsxitem name="doThis" title="Do Something">
      <jsxmenu name="subMenu" title="Submenu">
        <jsxitem name="toggle" title="Toggle Check for Subitem 2">
          <jsxitem name="sub2" title="Subitem">
            </jsxmenu>
          <jsxitem name="doThat" title="Do Something Else">
            </jsxitem>
          </jsxmenu>
        </jsxmenu>
      </jsxmenubar>

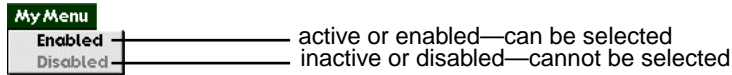
function menuSignal(menuItem)
{
  switch (menuItem.name)
  {
    case "doThis":alert("You chose the Do Something item.");
                  break;
    case "doThat":alert("You chose the Do Something Else item.");
                  break;
    case "toggle":subMenu.sub2.checked = !subMenu.sub2.checked;
                  break;
    default:alert ("Something went wrong...");
  }
}
```

When the user chooses a menu item, GoLive passes the associated `jsxmenuItem` object to the `menuSignal` function defined by the extension that defined the menu item. If the value of the object's `name` property is `toggle`, this code retrieves the menu item by name from the global namespace and sets the value of its `checked` property to the opposite of its current value.

Setting a Menu Item's Enabled State Explicitly

You can set a `jsxitem` object's `enabled` property to specify whether the menu item is enabled or disabled. [Figure 2.4](#) depicts an enabled menu item, which is drawn in black and can be chosen by the user. It also illustrates a disabled menu item, which is drawn in gray and cannot be chosen by the user.

FIGURE 2.4 Enabled and disabled menu items



The following code example creates the menu items that [Figure 2.4](#) depicts. This example sets each `jsxmenuItem` object's `enabled` property explicitly to enable or disable the menu item it defines.

```
<!-- Enable/Disable Example ----->

<jsxmenubar>
  <jsxmenu title="My Menu">
    <jsxitem name="enableItem" value="Enabled">
    <jsxitem name="disableItem" value="Disabled">
  </jsxmenu>
</jsxmenubar>
<script>
  <!-- //hide from old browsers

    MyMenu.disableItem.enabled = false;

// End of hiding scripts from old browsers-->
</script>
```

As you can see by the boldfaced lines above, this script simply accesses the Disabled menu item in the global namespace to set its `enabled` property. Because GoLive normally enables menu items by default, you need not set a menu item's `enabled` property to ensure that it is enabled. (Unless, of course, you've disabled it previously.)

In order to show clearly the techniques for enabling and disabling menu items, the preceding examples are somewhat oversimplified. Usually, you'll condition a menu item's `enabled` state on some prerequisite; for example, the **File>Close Window** menu item in GoLive is enabled only when a GoLive document window is open. Similarly, you'll condition a menu item's `checked` state on whether the behavior it provides is currently in effect; for example, the Window menu in GoLive places check marks next to the names of windows that are currently open. The next section shows how you can make GoLive set the states of your menu items automatically according to criteria you specify.

For information on testing for the presence of a document or its elements, see [Chapter 5, "Manipulating Document Objects."](#)

Setting The State of A Menu Item Automatically

The valueless dynamic attribute of the `<jsxitem>` tag marks its menu item as one that GoLive must initialize before displaying the menu. Each time the user attempts to open a menu defined by an extension, GoLive passes to the extension's `menuSetup` function each of the menu's items that have the dynamic attribute. The `menuSetup` function can then initialize the item for display, setting, for example, its `checked` state, its `enabled` state, and its initial value.

The following example marks both menu items as requiring dynamic initialization.

```
<jsxmenubar>
  <jsxmenu name="BGColr" title="Background">
    <jsxitem name="bgcolred" title="Set BG Red" dynamic>
    <jsxitem name="bgcolgrn" title="Set BG Green" dynamic>
  </jsxmenu>
</jsxmenubar>
```

The next example shows one way you might implement the `menuSetup` function for the extension that defines these menu items. This function sets the enabled and checked states of the menu item passed to it according to whether the current document's background color is red, green, or some other color. Like the `menuSignal` function discussed earlier in this chapter, the `menuSetup` function uses the passed menu item's name attribute as the case expression of a switch statement that customizes the function's actions according to the particular menu item passed to it.

```
function menuSetup(menuItem) {
    // begin with menu item unchecked
    menuItem.checked = false;
    // disable dynamic menu items if no document or no element
    menuItem.enabled = (document != null && document.element != null);
    // set enabled & checked states according to background color
    var tree = document.element;
    var bodyElement = tree.getSubElement("body");
    switch (menuItem.name) {
    case "bgcolred":
        if (bodyElement)
        {
            // allow user to change color to red if isn't red now
            menuItem.enabled = (bodyElement.bgcolor != "red")
            // place checkmark next to menu item if bgcolor is red now
            menuItem.checked = (bodyElement.bgcolor == "red")
        }
        break;
    case "bgcolgrn":
        if (bodyElement)
        {
            // allow user to change color to green if isn't green now
            menuItem.enabled = (bodyElement.bgcolor != "green")
            // place checkmark next to menu item if bgcolor is green now
            menuItem.checked = (bodyElement.bgcolor == "green")
        }
        break;
    }
}
```

Most of this code simply repeats techniques used earlier in this chapter; however, the code that tests the current background color of the active document deserves further scrutiny.

When GoLive interprets a document, it generates a tree of markup elements that correspond to the HTML elements the document defines. Each markup element takes the form of a set of C++ objects GoLive maintains internally, as well as a set of objects that make this internal

representation available in the JavaScript environment. To modify the active document, you use the properties and methods of JavaScript objects that the markup tree provides.

The global `document` object represents the currently-active GoLive document window. The `element` property of this object is a reference to the root of the markup tree GoLive generated when it read this document. Thus, the following code retrieves the active document's markup elements and stores them in the `tree` local variable.

```
var tree = document.element;
```

The `tree` variable actually holds the root of the markup tree, which is a special element intended for use only by GoLive itself. The markup elements defined by the document's HTML code—such as its head and body elements—are subelements of this root object. To modify the background color of the document, you must access the document's body element.

Each markup element provides a `getSubElement` method you can use to retrieve a particular element by name; thus, to retrieve the document's body element, simply pass its name to the `getSubElement` method of the `tree` object, as the following code does.

```
var bodyElement = tree.getSubElement("body");
```

The markup tree provides programmatic access to virtually every element of an active document; later on in this book, the [Custom Elements](#) and [Manipulating Document Objects](#) chapters discuss this subject in detail.

Adding Items to the Special Menu

To append your own menu items to the **Special** menu, place your `<jsxitem>` tags inside a `<jsxmenu>` element that specifies a value of "special" for its name attribute, as the following example does.

```
<jsxmodule name="DynamicMenus">
<jsxmenubar>
  <jsxmenu name="special">
    <jsxitem name="one" title="Special One" dynamic>
    <jsxitem name="two" title="Special Two">
    <jsxitem name="three" title="Special Three">
  </jsxmenu>
</jsxmenubar>
```

When GoLive adds multiple custom items to the **Special** menu, their order in the menu reflects the order in which GoLive loaded the modules that define the menu items, as well as the order in which each extension's `Main.html` file defines them when read top to bottom. You cannot control the order in which GoLive loads modules, but your menu items will always appear in the order that the `Main.html` file defines them.

As you can see in [Figure 2.5](#), the **Special One**, **Special Two**, and **Special Three** menu items this code defines appear at the bottom of the Special menu. Note the presence of the **XML Tree** and

Selections submenus, which were loaded first because the name of the module that defines them, DOM, precedes the name of the `DynamicMenus` module in alphabetic order.

FIGURE 2.5 Custom items appended to the Special menu



3

Dialogs and Palettes

The GoLive Extend Script SDK provides tags you can use to create two kinds of windows:

- A **modal window**, also known as a **modal dialog box**
 - Requires a response before allowing the user to proceed.
 - Does not provide a close box; instead it provides a control, such as a button, that can dismiss the dialog.

The alert dialogs shown in [Figure 2.2 on page 44](#) are examples of modal windows.

- A **modeless window**, also known as a **palette**
 - Floats above open document windows without requiring a user response.
 - Is never terminated but can be hidden.
 - Is hidden until the user selects it in the Window menu.

The Objects palette and the Inspector window are examples of modeless windows.

This chapter describes how to create these kinds of windows. Topics this chapter describes include

- [Defining the Modal Dialog Window](#)
- [Defining Dialog Content](#)
- [Displaying the Dialog](#)
- [Implementing the controlSignal Function](#)
- [Floating Palettes](#)

Modal Dialog Windows

This section describes the steps required to display a modal dialog window:

- [Defining the Modal Dialog Window](#)
- [Defining Dialog Content](#)
- [Displaying the Dialog](#)

Optionally, you can cause the dialog to respond to controls other than buttons that dismiss the dialog; for details, see [“Implementing the controlSignal Function” on page 60](#).

Defining the Modal Dialog Window

The `jsxdialog` element defines the modal dialog window itself. You'll use other tags to define window content such as text and buttons.

```
<jsxdialog name="objectName" title="Title On Windows OS" width="anInteger"
           height="anInteger" >
</jsxdialog>
```

This tag specifies the following attributes:

<i>name</i>	The name under which the dialog's JavaScript representation appears in the global namespace. GoLive also makes the dialog available via the <code>dialogs</code> array it maintains in the global namespace.
<i>title</i>	The dialog window's title when displayed on Windows platforms. Not used on Mac OS platforms. (Mac OS dialog boxes do not have titles.)
<i>width</i>	Width of the dialog; can be overridden by an embedded <code><table></code> tag.
<i>height</i>	Height of the dialog; can be overridden by an embedded <code><table></code> tag.

The element that defines an empty modal dialog window would look like the following example. This example is not meant to be a complete example of all the code required to create a dialog—it's just meant to show how the `<jsxdialog>` tag is used. The definition of any `<jsxdialog>` must always include at least one `<jsxcontrol>` that can dismiss the dialog; otherwise, the user cannot exit the dialog.

```
<jsxdialog name="testDialog" title="Test Dialog" width="215" height="200">
  // jsxcontrol tags defining content such as text & controls go here
</jsxdialog>
```

IMPORTANT: *Every `jsxdialog` element must contain at least one `jsxcontrol` element that can dismiss the modal dialog; otherwise, the user cannot exit the dialog.*

The next section describes the use of the `<jsxcontrol>` tag to define the content of the modal dialog, including the all-important control that can dismiss the dialog.

Defining Dialog Content

The `<jsxcontrol>` tag can define a variety of control objects—the value of its `type` attribute specifies whether it defines a pushbutton, a checkbox, a radio button, various text fields, various popup menus, or a custom control of your own design.

```
<jsxcontrol type="KindOfControl" name="JavaScriptName"
            value="InitialValue" posx="NumOfPixels" posy="NumOfPixels"
            width="NumOfPixels" height="NumOfPixels"
            halign="SeeReference" valign="SeeReference" >
```

All attributes other than `halign` and `valign` are required.

All `jsxcontrol` objects provide certain common behaviors, such as the ability to draw themselves in the location you specify. They also have certain common attributes, such as those which specify the control's position. Any of your `<jsxcontrol>` tags can define these attributes, and your JavaScript code can get or set their corresponding properties in the `jsxcontrol` objects these tags create.

Each type of control object also provides its own specialized attributes and functions that other types of control objects don't provide; for example, text-entry fields can capture keystrokes, but radio buttons cannot. If your JavaScript code tries to set the value of an attribute or property that does not apply to the control it is trying to set, the control ignores the non-applicable input. For example, if you write a JavaScript statement that tries to set the `itemCount` property of a radio button, the control ignores the statement because radio buttons have no such property. Similarly, if your script tries to call a function that the control does not supply, the control ignores the statement. For example, if you try to call the `addItem` function of any control other than a popup menu, GoLive ignores the `addItem` function call.

The value of the `type` attribute must be one of those listed in [jsxcontrol on page 122](#). As mentioned previously, the `type` attribute specifies the kind of control this tag creates; thus, each `type` value specifies a particular customized appearance, behavior, and set of JavaScript properties and functions for the `jsxcontrol` object that GoLive creates as the result of interpreting the `<jsxcontrol>` tag. For example, a `jsxcontrol` of type `button` provides the appearance and behaviors of a pushbutton.

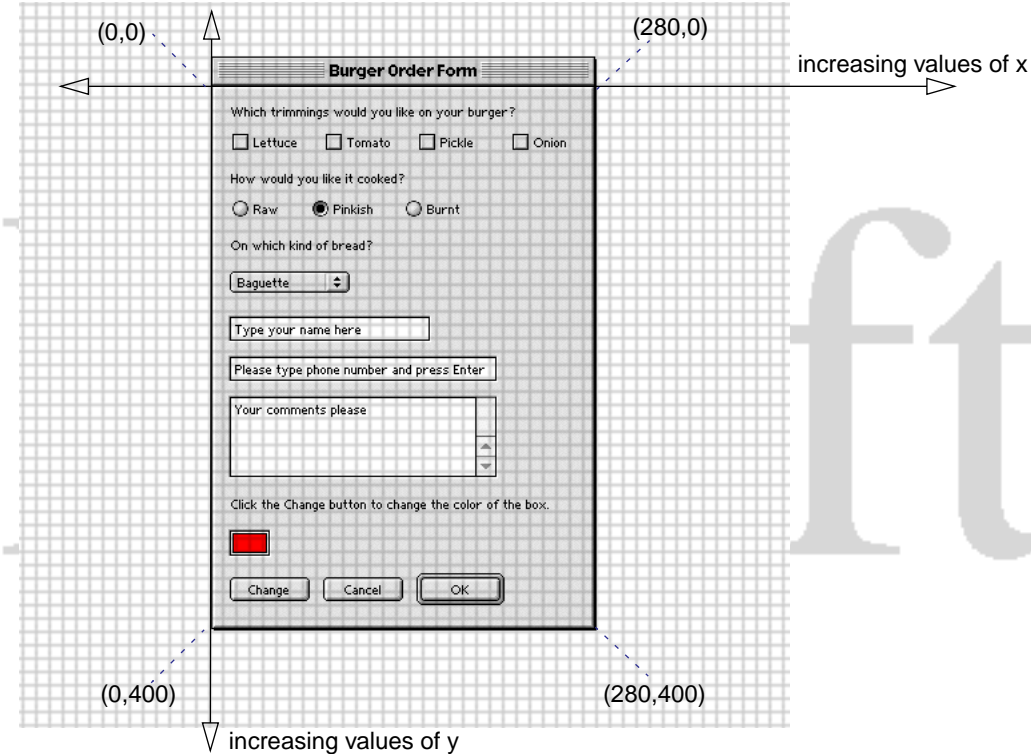
A modal dialog does not provide a close box; to close a modal dialog window, you must implement one or more controls that can dismiss the dialog. The control that dismisses the dialog does not need to be a pushbutton. The SDK provides two ways to dismiss a dialog:

- Any control created with a `name` attribute of `dialogok`, `dialogcancel`, or `dialogother` closes the dialog when the user clicks it. A control having any other value as its `name` attribute does not dismiss its dialog.
- The `exitModal` method of the [Dialog Object](#) dismisses the dialog unconditionally. The argument to this method specifies the value that the `runModal` method returns when it exits.

The `value` attribute specifies only the initial value of the object a `<jsxcontrol>` tag creates, not the final value the control returns to GoLive when the user changes the control's state. A `pushbutton` control's final value is always the same as its initial value, but this is not always so for other control types, such as popup menus.

The `posx` and `posy` attributes specify the control's placement in the modal dialog. These attributes define an (x,y) coordinate pair that specifies the location of the control's upper-left corner within the coordinate plane having its origin at the upper-left corner of the dialog. The values of `posx` and `posy` are specified in pixels.

FIGURE 3.1 Positioning controls in dialog's coordinate plane



The `width` and `height` attributes specify the dimensions of the control in pixels.

The optional `halign` and `valign` attributes specify how the control positions itself when its containing window is resized; because the user cannot resize a modal dialog, this section does not discuss these attributes.

Creating A Cancel Button

A `jsxcontrol` of type `button` with name `dialogcancel` is a `pushbutton` that closes its dialog. The following example uses the `<jsxcontrol>` tag to define a **Cancel** button resembling the one [Figure 3.2](#) depicts.

```
<jsxcontrol type="Button" name="dialogcancel" value="Cancel"
            posx="80" posy="138" width="60" height="18">
</jsxcontrol>
```


Creating An OK Button

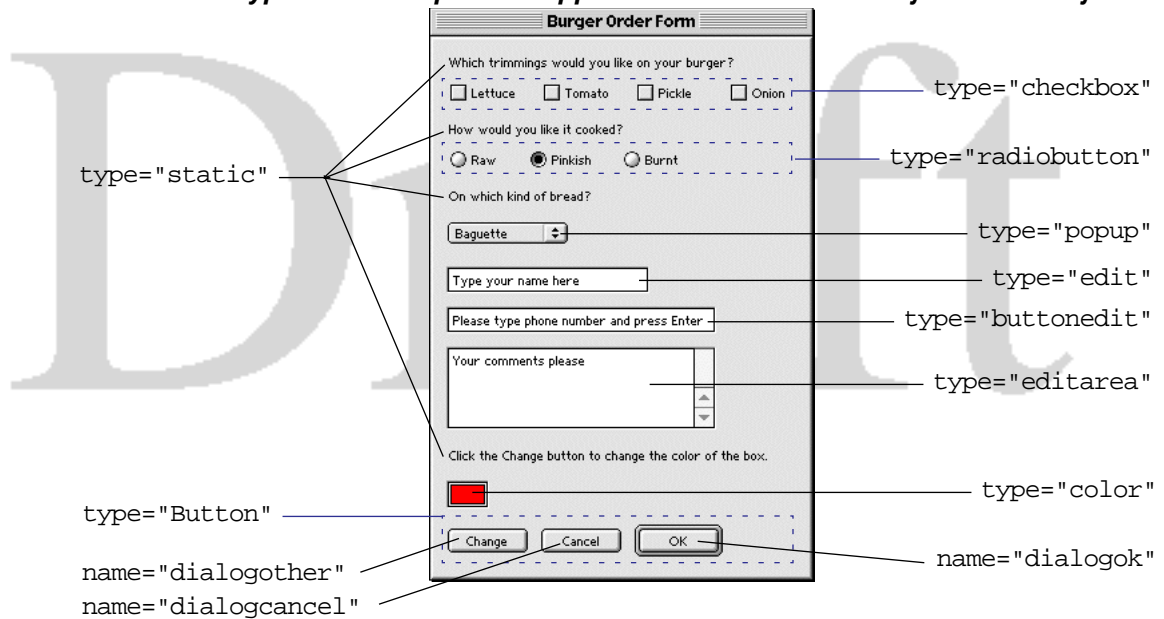
The following HTML example uses the `<jsxcontrol>` tag to define an **OK** button. GoLive recognizes a `jsxcontrol` of type `button` with name `dialogok` as a button that closes the dialog when the user clicks it. Additionally, this button has a special outlined appearance that identifies it as

- the default choice in the dialog
- the button that is clicked when the user presses either of the Return or Enter keys.

```
<jsxcontrol type="Button" name="dialogok" value="OK"
           posx="80" posy="138" width="60" height="18">
</jsxcontrol>
```

When GoLive interprets the `jsxcontrol` tag in this example, it creates a button resembling the **OK** button in [Figure 3.2](#).

FIGURE 3.2 Type attribute specifies appearance and behavior of `jsxcontrol` object



Creating Other Kinds of Controls

The following example illustrates the use of the `<jsxcontrol>` tag to create the static text, editable text, checkboxes, radio buttons, a popup menu, and color picker shown in the dialog box [Figure 3.2](#) depicts. For many additional examples of the use of controls, see the sample code that the SDK provides.

```
// example of using <jsxcontrol> to create various kinds of controls
<jsxdialog name="myModalDialog" title="Burger Order Form" width="280" height="400" >
  <jsxcontrol type="static" name="trimmingsPrompt"
    value="Which trimmings would you like on your burger?"
    posx="10" posy="10" width="240" height="18">
  <jsxcontrol type="checkbox" name="lettuceBox" value="Lettuce"
    posx="10" posy="30" width="60" height="18">
  <jsxcontrol type="checkbox" name="tomatoBox" value="Tomato"
    posx="80" posy="30" width="60" height="18">
  <jsxcontrol type="checkbox" name="pickleBox" value="Pickle"
    posx="150" posy="30" width="60" height="18">
  <jsxcontrol type="checkbox" name="onionBox" value="Onion"
    posx="220" posy="30" width="60" height="18">
  <jsxcontrol type="static" name="cookingPrompt"
    value="How would you like it cooked?"
    posx="10" posy="60" width="200" height="50">
  <jsxcontrol type="radiobutton" name="rareRadio" value="Raw"
    posx="10" posy="80" width="60" height="18" group = "cookBtn">
  <jsxcontrol type="radiobutton" name="medRadio" value="Pinkish"
    posx="70" posy="80" width="60" height="18" group = "cookBtn">
  <jsxcontrol type="radiobutton" name="wellRadio" value="Burnt"
    posx="140" posy="80" width="60" height="18" group = "cookBtn">
  <jsxcontrol type="static" name="bunPrompt" value="On which kind of bread?"
    posx="10" posy="110" width="300" height="18">
  <jsxcontrol type="popup" name="breadMenu"
    value="Baguette, Pita, Rye, Sourdough, Wheat"
    posx="10" posy="135" width="90" height="18">
  <jsxcontrol type="edit" name="custNameField" value="Type your name here"
    posx="10" posy="170" width="150" height="18">
  <jsxcontrol type="buttonedit" name="phoneField"
    value="Please type phone number and press Enter"
    posx="10" posy="200" width="200" height="18">
  <jsxcontrol type="editarea" name="commentField" value="Your comments please"
    posx="10" posy="230" width="200" height="60">
  <jsxcontrol type="static" name="chipsPrompt"
    value="Click the Change button to change the color of the box."
    posx="10" posy="304" width="300" height="18">
  <jsxcontrol type="color" name="color" posx="10" posy="330" width=30 height=19>
  <jsxcontrol type="Button" name="dialogOther" value="Change"
    posx="10" posy="365" width="60" height="18">
  <jsxcontrol type="Button" name="dialogcancel" value="Cancel"
    posx="80" posy="365" width="60" height="18">
  <jsxcontrol type="Button" name="dialogok" value="OK"
    posx="153" posy="365" width="60" height="18">
</jsxdialog>
```

Displaying the Dialog

To display a modal dialog, call the `jsxdialog` object's `runModal` function. When the `runModal` function completes, it returns one of the following values:

<i>dialogcancel</i>	0	User clicked the Cancel button.
<i>dialogok</i>	1	User clicked the OK button.
<i>dialogother</i>	2	User interacted with a control other than the OK or Cancel buttons.

Your extension uses this return value to take appropriate action; for example, if the `runModal` function returns the `dialogok` value, your extension would accept the current values of the controls in the dialog as valid user input. On the other hand, if the `runModal` function returns the `dialogcancel` value, your extension would discard the user's input to the dialog.

The following example shows different ways to use the `runModal` function's result.

```
function MenuSignal(menuItem){
    switch(menuItem.name){
        // how to call runModal for simple Cancel/OK dialog
        case "mySimpleDialog":{
            if (myOtherDialog.runModal())
                alert("That was a wise decision!"); break;
            else
                alert ("Are you sure you want to cancel?"); break;
        }
        // how to call runModal for multi-button dialog
        case "myRegDialog":
            { // shareware registration dialog example
                switch (myRegDialog.runModal()) {
                    case "dialogok":
                        // accept user input
                        var userName = myDialog.myNameField.value;
                        var credit = myDialog.myCreditCardField.value;
                        break;
                    case "dialogcancel":
                        // restore defaults
                        myDialog.myNameField.value = "";
                        myDialog.myCreditCardField.value = "";
                        break;
                    case "dialogother":
                        // a third alternative
                        myDialog.myNameField.text = "Demo User";
                        myDialog.myCreditCardField.value = "DEMO DEMO DEMO DEMO";
                        setUpAsDemoVersion();
                        break;
                } // end switch
            } // end myRegDialog case
    }
}
```

For a dialog that allows the user only to accept or reject the choice it offers, you can use the straightforward approach that the `mySimpleDialog` case takes. Because `dialogCancel` actually is the value zero, you can treat non-zero results from the `runModal` function as the boolean value `true`.

For a dialog that offers several exit points, you'll need to take the approach demonstrated by the `myRegDialog` case—it uses the `runModal` function's return value as the case expression in a `switch` statement that customizes its actions according to the button that dismissed the dialog.

After the dialog closes, all of its `jsxcontrol` objects remain available in the JavaScript namespace; as a result, retrieving the current control settings or restoring default values is straightforward.

Implementing the `controlSignal` Function

If the only information GoLive needs from your dialog is whether the user accepts or rejects the information it displays, you don't need to use the `controlSignal` function. You can determine whether the user clicked OK, Cancel, or something else by examining the value that the `runModal` function returns. However, if you need to respond to changes in the state of a `jsxcontrol` object while a modal or modeless dialog is displayed, you can use the optional `controlSignal` function to do so; for example, you would use this function to update the appearance of an element in response to changes in the value of a control.

Whenever the user changes the state of a control created by the `jsxcontrol` tag, GoLive passes the control to the `controlSignal` function of the extension that displayed the control. Your implementation of this optional function takes any action necessary to respond to the change in the control's state.

Typically, this function tests the `name` attribute of the control passed as its argument, using `control.name` as the case expression of a `switch` statement that customizes this function's actions for various controls.

Here's a simple example of a `controlSignal` function that you could use to add functionality to the dialog shown in [Figure 3.2](#). When the user clicks the Change button, this `controlSignal` function's `dialogOther` case changes the color of the `color` control. For all

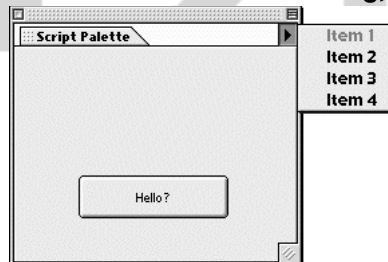
other non-dismissing controls, the default case prints an informative statement to the **Javascript Output** palette.

```
function controlSignal(control)
{
    switch (control.name)
    {
        case "dialogOther": var currColor = myModalDialog.color.value
                            if (currColor == "red")
                                myModalDialog.color.value="green"
                            else
                                myModalDialog.color.value="red"
                            break;
        case "aNother":// each case responds to a specific control
                        break;
        default:writeln (control.name + " selected.");
    }
}
```

Floating Palettes

The `<jspxpalette>` tag creates a floating window with its own pull-down menu, like the one [Figure 3.3](#) shows.

FIGURE 3.3 *Modeless dialog, floating window, or palette*



The palette window's title appears in the **Window** menu; for example, the palette that [Figure 3.3](#) depicts places the **Script Palette** item in the **Window** menu. A palette window always runs while GoLive is running; when the user clicks its close box, GoLive hides the palette from the user but the palette actually continues to run.

A `jspxpalette` element looks like the following example.

```
<jspxpalette name="objectName" title="TitleOfPalette" order="anInteger"
              width="anInteger" height="anInteger" >
    // for palette menu, add <jsxmenu> and <jsxitem> elements here
</jspxpalette>
```

The name, width, and height attributes to the `<jspxpalette>` opening tag act as their counterparts in other tags do.

The `title` attribute defines the palette's title as it appears in the Window menu and on the tab of the palette window.

The `order` attribute determines the position in which this palette's `title` attribute appears in the Window menu. Higher values of the `order` attribute place the palette's `title` closer to the bottom of the Window menu in the GoLive design environment. Values greater than 9999 produce undefined results.

The following example creates the palette shown in [Figure 3.3](#).

```
<!-- Palette ----->
// menuorder is at the end of the Objects group (0101)
<jsxpalette name="JSXPalette" title="Script Palette" order=0151
    width="215" height="164">
    <jsxcontrol type="custom" name="custom" posx="10" posy="10"
        width="112" height="32">
    <jsxcontrol type="Button" name="button" value="Hello?"
        posx="48" posy="96" width="112" height="32">

<!-- Palette Menu ----->

    <jsxmenu name="firstMenu" title="test">
        <jsxitem name="item1" title="Item 1" dynamic>
        <jsxitem name="item2" title="Item 2">
        <jsxitem name="item3" title="Item 3">
        <jsxitem name="item4" title="Item 4">
    </jsxmenu>
</jsxpalette>
```

As noted previously, the `name`, `height`, and `width` parameters behave as they do for all the other tags the GoLive Extend Script SDK provides, so this section won't discuss them. For examples of the `title` attribute's appearance in the palette and in the Window menu, see [Figures 3.3](#) and [3.4](#), respectively.

The `order` attribute defines a sort order for this palette among the other items and groups of items that appear in the Window menu. The first two digits of this value specify the group of menu items amongst which this item appears, while the second two digits define this item's sort order within the group.

[Appendix B, "Sort Order Tables"](#) lists the values GoLive uses to sort built-in items that always appear in the Window menu. According to this table, the Objects and Color menu items have sort values of 0101 and 0110, respectively. Thus, the **Script Palette** palette's `menu` attribute of 0151 specifies that it is to appear in after the Color item, as [Figure 3.4](#) shows.

FIGURE 3.4 Sort order in Window menu



Palette Menus

To add a palette menu to a `jsxpalette`, simply add a `jsxmenu` element and its associated `jsxitem` elements to the body of your `jsxpalette` definition, as the code example at the beginning of this section does. You cannot define more than one menu for each palette, but that palette menu can define submenus.

Other Controls

A `jsxpalette` window can host any of the `jsxcontrol` objects that the SDK provides. These controls behave the same way in a floating palette as they do in a running modal dialog, with one exception: pushbuttons named `dialogOK`, `dialogCancel`, or `dialogOther` do not close or hide palette windows.

Using the Dialog Editor Extension

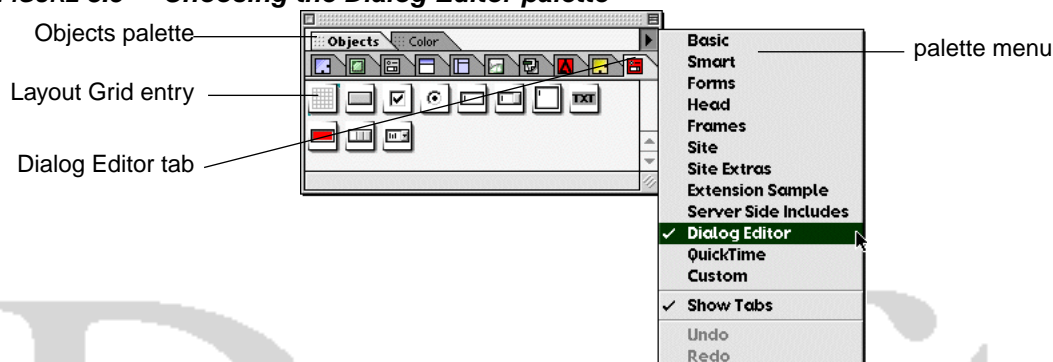
The Dialog Editor sample extension provides a user interface you can use to design `jsxcontrol` objects visually. Its output is intended for use in the GoLive design environment, not in Web browsers.

NOTE: The Dialog Editor generates only the source code that defines a dialog's appearance. You must incorporate the output of the Dialog Editor into a `Main.html` file which provides the additional code required to display and dismiss the dialog.

You can take the following steps to use the Dialog Editor extension:

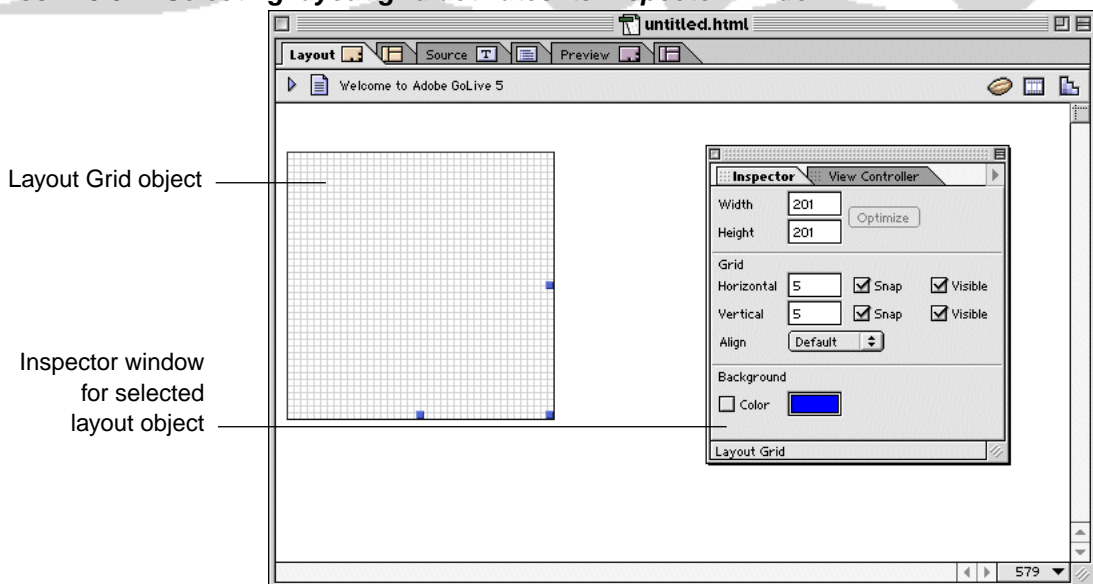
1. Create a new, empty GoLive document window by choosing the **File>New** menu item.
2. Display the Dialog Editor's palette entries by clicking its tab in the **Objects** palette or choosing the **Dialog Editor** item from the Objects palette's palette menu, as [Figure 3.5](#) depicts.

FIGURE 3.5 Choosing the Dialog Editor palette



3. Whenever you use the **Dialog Editor** extension to create a dialog, you must begin by creating a layout grid that will contain the other controls in the dialog. To do so, drag the **Layout Grid** palette entry onto the GoLive document window. A layout grid object similar to the one in [Figure 3.6](#) appears.

FIGURE 3.6 Selecting layout grid activates its inspector window

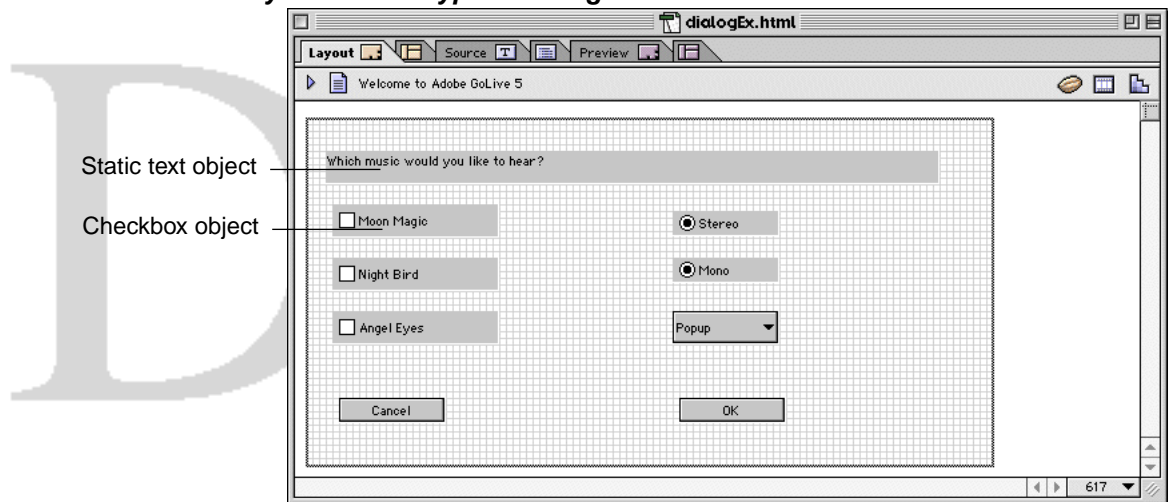


4. In the document window, click the layout grid object once to select it. When this object is selected, a border and handles appear, and the Inspector window displays the layout grid's properties, as [Figure 3.6](#) shows.

5. The layout grid object is a substitute for the dialog window that will hold your dialog's content. Drag the handles or use the controls in the Inspector window to set the size and position of the layout grid object to match that of the dialog window that will display the dialog's content.
6. Define your dialog's content:
 - Drag an icon from the Dialog Editor palette onto the layout grid object in the document window.
 - GoLive creates a new element and displays its inspector window.
 - Set the new element's `name` attribute to a unique value.

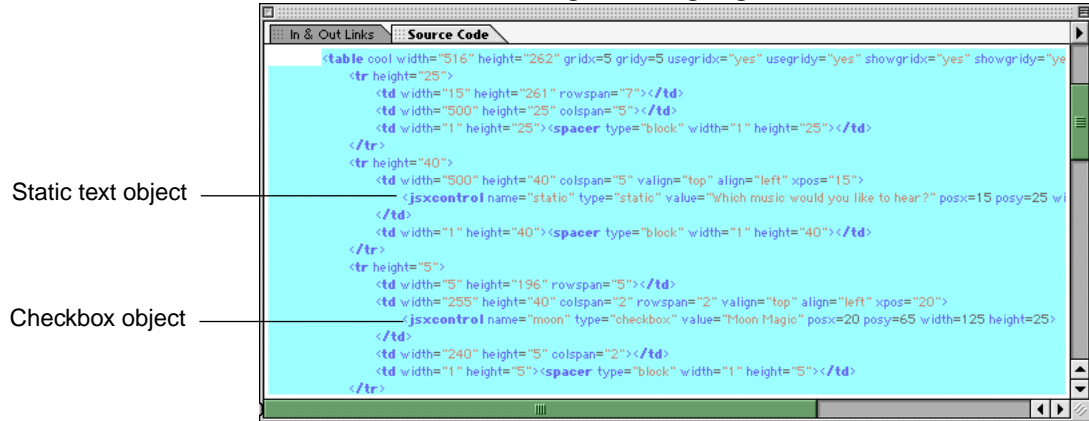
IMPORTANT: *To ensure reliable access to JavaScript objects, each `name` attribute must be unique within the JavaScript namespace.*

FIGURE 3.7 Layout view of typical dialog content



7. When you've finished defining your dialog, select it by clicking the layout grid once.
8. Choose the **Window>Source Code** menu item to display a **Source Code** window similar to the one in [Figure 3.8](#). When the layout grid is selected, the **Source Code** window highlights the code that defines the dialog.

FIGURE 3.8 Source Code window with dialog code highlighted



9. Copy each `jsxcontrol` element from the highlighted code and paste it into the appropriate container element in the destination `Main.html` file:
 - To create a modeless dialog, paste the `jsxcontrol` elements into a `jspxpalette` element. For a code example, see [“Floating Palettes” on page 61](#).
 - To create a modal dialog, paste the `jsxcontrol` elements into a `jsxdialog` element. For a code example, see [“Defining Dialog Content” on page 55](#).
10. In the `Main.html` file, implement code that displays and dismisses the dialog:
 - For modal dialogs,
 - Call the dialog’s `runModal` method to display the dialog.
 - Call the dialog’s `exitModal` method to dismiss the dialog.Alternatively, you can provide a control having a name attribute `dialogok`, `dialogcancel`, or `dialogother`; when clicked, such controls dismiss the modal dialog. For more information, see [“Defining Dialog Content” on page 55](#).
 - For modeless dialogs, you need not provide any additional code to display or dismiss the dialog.
 - The `jspxpalette` element installs your palette in the **Windows** menu.
 - All `jspxpalette` objects provide a close box that the user can click to dismiss the modeless dialog.

4

Custom Elements

The user can drag icons from the Objects palette to a GoLive document window to add predefined elements to the page. This chapter describes how to define a custom HTML element and install it in the Objects palette as an icon the user can drop into a GoLive document.

Overview

When the user drags an icon from the Objects palette to a GoLive document window, GoLive

- adds the icon's associated markup tags to the document
- places in the GoLive document window a visual representation of the content the icon provides.

This visual representation, called a **custom box**, can reflect the actual appearance of the custom element or it can draw a placeholder graphic; for example, a placeholder graphic could be used to represent server-side content not available at the time the icon is dropped onto the document window.

A custom box is like a GoLive **component** in the sense that it adds multiple elements to the page. However, the content of a component is limited to that which can be created with standard HTML tags, and the component offers no JavaScript access to its content.


In contrast, a custom box

- Defines a custom tag name that represents it in the GoLive design environment; thus you can use custom boxes to define elements created from any mixture of your own custom tags, the SDK provides, and standard HTML tags.
- Provides JavaScript access to the attributes of the custom element it represents; thus, the custom box enables you to inspect what you might otherwise create as a component.

Tags For Creating Custom Elements

You'll use the following tags to create custom entries in the Objects palette:

- The `<jsxpalettegroup>` tag creates a new tab inside the palette (or selects an existing tab) which is then filled in with a number of `<jsxpalettentry>` tags.
- Each of the `<jsxpalettentry>` tags is a placeholder for the HTML code the palette entry adds to the page; usually, this code contains a custom tag defined by the `<jsxelement>` tag.
- The `<jsxinspector>` tag is used to create a dialog which serves as the inspector for that specific box. The necessary icons for the palette as well as necessary pictures to draw the box may be defined using the widely known `` tag.



Each of these tags has a `classid` attribute that GoLive uses to identify the elements of the custom box. All of the tags used to define a custom box must specify the same `classid` attribute value. This value identifies the palette entry that creates the custom box and inserts HTML in the document, as well as the inspector window to display when the box is activated. This value must be unique within the extension module.

Custom Box Event-Handling Functions

When the user drags a custom palette entry's icon into a GoLive document window, GoLive creates an empty custom box and places it in the document. Subsequently, it calls the following functions your extension must provide:

- To draw the visual representation of the palette entry, GoLive calls the [drawBox](#) function, passing the Box object to be drawn as its argument. This function may use the global Draw object to draw its contents.
- When the user resizes the box by dragging at the box's borders or by entering data into the inspector dialog, GoLive calls the extension's [boxResized](#) and [parseBox](#) functions.
 - The `boxResized` method accepts three parameters: the box object itself, the new width, and the new height. In addition to updating the box's appearance to match the new size, your implementation of this method must also update the markup elements that the box represents.
 - After calling the `boxResized` method, GoLive calls the [parseBox](#) method, passing the box object as its argument. This method adjusts the appearance of the box according to the properties of the markup elements the box represents.
- GoLive also calls the extension's [parseBox](#) method
 - when a document containing the palette entry is read.
 - when the user switches to Layout View from another view in the document window.

Development Overview

The rest of this chapter describes the following steps required to create an Objects palette entry that provides a custom box:

- Define the custom element itself
 - Define a custom tag name
 - Define the custom tag's HTML content
- Install the custom element as an icon in the Objects palette
- Implement functions that enable interaction with an instance of the custom element in a GoLive document window.

Defining A Custom HTML Tag

You can use the [jsxelement](#) tag to define the name of a custom HTML tag in the GoLive environment. The content this custom tag adds to the page is defined elsewhere, in the [jsxpalettentry](#) tag that is a companion to this [jsxelement](#) tag.

```
<jsxelement tagName="nameOfCustomTag" classid="yourUniqueID"
              type="kindOfElement">
</jsxelement>
```

The `tagName` attribute of the `<jsxelement>` opening tag specifies the name of your custom HTML tag; for example, a `tagName` attribute of `mytag` defines the `<mytag>` custom tag in the GoLive environment. If this name duplicates that of any existing markup tag or JavaScript object, your custom tag replaces the built-in tag and disables the built-in inspector for that tag, so you should choose the name of your tag with this in mind.

The `jsxelement` tag and its associated `jsxpalettentry` tag must specify identical `classid` attributes. The `classid` attribute is an identifier that associates a `jsxpalettentry` element and a `jsxinspector` element with this `jsxelement` element. The value of a `classid` attribute can be any text string that is not already used as an identifier elsewhere in the extension.

NOTE: The `classid` attribute does not provide any sort of built-in behavior or data. It's just an identifier GoLive uses to gather up all the code items it expects a custom element to provide. The `classid` attribute is not a class name in the classic object-oriented programming sense, and it has no relationship to the class names of objects that GoLive uses internally.

The `type` attribute specifies the kind of element this tag defines, such as a container, a server-side include, or some other kind of HTML element; thus, the value of this attribute defines certain aspects of the element's behavior and appearance. For a complete list of the kinds of elements the `jsxelement` tag can define, see the description of the `type` attribute in the [jsxelement](#) reference entry on [page 126](#).

The following example defines the `<mytag>` tag.

```
<jsxelement tagName="mytag" classid="myclass" type="plain">
```

The `tagname` attribute of `mytag` specifies that this `jsxelement` tag defines the `<mytag>` custom tag. Whether `<mytag>` is binary (requires a `</mytag>` closing tag) or unary (no closing tag) is defined elsewhere, in the `jsxpalettentry` having the same `classid` value.

The `classid` attribute of `myclass` matches the `classid` attribute of the `<jsxpalettentry>` and `<jsxinspector>` tags associated with this `<jsxelement>` tag.

The `type` attribute of `plain` specifies that this is a standard HTML tag, rather than a specialty tag such as a container.

Do not attempt to define your custom tag's custom attributes here; instead, use the `jsxpalettentry` tag to define them. The next section describes how to use this tag to define the custom element's HTML content and install it in the Objects palette.

Defining the Custom Tag's Palette Icon and HTML Content

The [jspxpalettentry](#) tag associates two important things with the custom tag that its associated [jsxelement](#) tag defines:

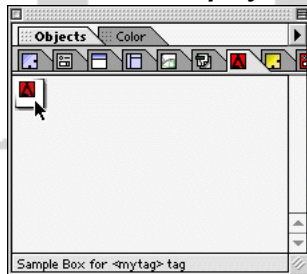
- The HTML code this tag adds to the page when the user drops its icon in a GoLive document.
- The icon that represents the custom tag in a [jspxpalettegroup](#) tab in the Objects palette.

The syntax of the `<jspxpalettentry>` tag is as follows.

```
<jspxpalettentry
  display="DescriptionOfTag" classid="yourUniqueID"
  picture="paletteIcon"
    < customTagName attribute1 attribute2 ... attributeN > // custom elt
      optional additional elements here
    </customTagName> // optional
</jspxpalettentry>
```

The `display` attribute holds a short description of the custom element this palette item represents. [Figure 4.1](#) depicts an example of this description, which appears in the lower-left corner of the Objects palette when the mouse cursor pauses over the custom icon.

FIGURE 4.1 *Display attribute of `jspxpalettentry`*



The `classid` attribute specifies a unique identifier that associates this `jspxpalettentry` tag with its companion `jsxelement` and `jsxinspector` tags. The palette entry's `classid` attribute must match the `classid` attributes of its associated `jsxelement` and `jsxinspector` tags exactly.

The `picture` attribute specifies the picture used as this custom element's Objects palette icon. GoLive scales this picture to 24 pixels high by 24 pixels wide automatically when it installs the palette entry's icon. The value of the `picture` attribute is the name attribute of an `` tag defined elsewhere in the extension's `Main.html` file. This `` tag's `src` attribute specifies the path to the picture to use as the custom element's icon.

Following the `picture` attribute is the HTML code that the custom element adds to the generated page:

- The custom tag and its attributes

This HTML must begin with a left bracket (`<`) followed by the tag name the associated `jsxelement` tag defines. Following that are the custom tag's attributes. Finish the definition of the custom HTML content with a closing bracket (`>`)

- The HTML code this tag is to add to the page.

Your definition of this code can be composed of any mixture of standard HTML tags, special tags provided by the SDK, or your own custom tags. When the element is dropped on the page, GoLive parses its tags and generates the HTML it adds to the document. In this generated HTML,

- `<jsx...>` tags provided by the Extend Script SDK are replaced by standard HTML with additional custom attributes.
- custom elements defined by `<jsxelement>` tags are added to the GoLive document's HTML source exactly as defined.

- An optional closing tag that takes the form of the custom tag with a forward slash inserted between the leftmost bracket and the tagname; for example the closing tag for the `<mytag>` tag is the `</mytag>` tag.

Here is an example `jsxpalettentry` element.

```
<jsxpalettentry display="Sample Box for <test> tag"
                classid="test" picture="paletteIcon">
  // this palette entry places two custom elements and an image on the page
  <test name="myFirst" width=100 height=50 src="http://www.adobe.com">
  
  <test name="myLast" width=100 height=50 src="http://www.adobe.com">
</jsxpalettentry>
```

NOTE: This example has been reformatted to enhance its readability. Do not include line breaks in the HTML code your `<jsxpalettentry>` element adds to the page. Line breaks in this code can cause cross-platform incompatibilities.

The `display` attribute of this palette entry produces the display text shown in the lower-left corner of the palette in [Figure 4.1](#).

The `classid` of `test` is a unique value shared only by the `classid` attributes of the `jsxelement` and `jsxinspector` tags associated with this palette entry.

The `picture` attribute value of `paletteIcon` is the name attribute of the following `` tag, defined elsewhere in this extension's `Main.html` file.

```

```

Following the `picture` attribute is the closing bracket that finishes the `<jsxpalettentry>` start tag.

Following the `<jsxpalettentry>` start tag is the HTML that this palette entry adds to the page. This palette entry places an image and two instances of the `test` custom element on the page. The `test` element consists of the `<test>` tag and the content it adds to the page—the name, width, height, and `src` attributes.


```
<test name="myFirst" width=100 height=50 src="http://www.adobe.com"><test name="myLast" width=100 height=50 src="http://www.adobe.com">
```

The mess above isn't a typographical error, it's how your custom element definition should look. Don't include line breaks in the definition of your custom element—they cause cross-platform incompatibilities in your code.

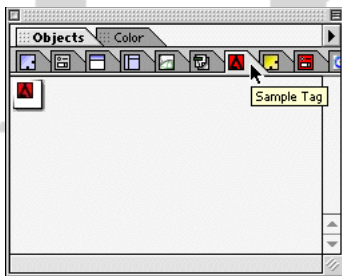
The attributes associated with your custom element are defined in the HTML the palette entry adds to the page. Note that this palette entry adds two `test` elements to the page, and that each `test` element can define and initialize its respective attributes as necessary to produce the desired output.

Finally, the `</jspxpalettentry>` end tag finishes the definition of the palette entry.

Installing A Custom Entry In the Objects Palette

The [jspxpalettegroup](#) element specifies the tab under which its `jspxpalettentry` items appear in the Objects palette. This element can be used to add entries to one of the built-in Objects palette tabs or it can be used to define a custom tab under which its entries appear; for example, [Figure 4.2](#) depicts the addition of the **Custom Box** sample extension's custom tab.

FIGURE 4.2 Custom tab in Objects palette.




The [jspxpalettegroup](#) element looks like the following example.

```
<jspxpalettegroup
  name="objectName" display="tabName" tabOrder="anInteger"
  picture="tabIcon" order=anInteger >
</jspxpalettegroup>
```

As usual, the `name` attribute specifies the name under which the tab's JavaScript representation appears in the global namespace.

- To add this group of palette entries to a custom tab, specify your own unique identifier as the value of this attribute.
- To add this group of palette entries to one of the built-in Objects palette tabs, specify one of the predefined name attributes described in the [jspxpalettegroup](#) section beginning on [page 125](#). If you use a predefined name attribute, omit the `display`, `picture`, and `taborder` attributes from your `jspxpalettegroup` tag—the built-in tabs provide these attributes for you and you cannot change them.



The `display` attribute specifies the text that appears when the mouse pointer pauses over the tab on the screen; for example, in [Figure 4.2](#), the tab's `display` attribute holds the Extension Sample value. If you use a predefined name attribute, your `jsxpalettegroup` tag can omit the `display` attribute.

The `picture` attribute specifies the icon that appears in the custom tab. The value of this attribute is the name attribute of an `` tag that specifies the path to this picture as the value of its `src` attribute. If you use a predefined name attribute, your `jsxpalettegroup` tag can omit the `picture` attribute.

The `taborder` attribute specifies the sort order for this tab amongst all tabs appearing in the **Objects** palette. **Objects** palette tabs are sorted by `taborder` value from least to greatest, with higher values appearing closer to the rightmost edge of the palette. If you use a predefined name attribute, your `jsxpalettegroup` tag can omit the `taborder` attribute.

The `order` attribute is used only to add palette entries to one of the built-in tabs, such as the **Basic** tab. When you add palette entries to a built-in tab, the value of the `order` attribute specifies the position at which GoLive adds all of your custom entries. Your icons are always added to the palette in the order your extension defines them. Thus, if the value of your `order` attribute that is less than that used by a built-in palette entry, your icons are put into the palette before that built-in icon.

Adding Palette Entries to a Built-in Tab

To add custom palette entries to one of the built-in **Objects** palette tabs, specify one of the predefined name attributes described in the [jsxpalettegroup](#) section beginning on [page 125](#); for example, the following `jsxpalettegroup` tag adds its `jsxpalettentry` icons to the **Basic** tab.

```
<jsxpalettegroup name="Basic"order=3000>
    //jsxpalettentry tags appearing here are added to the Basic tab
</jsxpalettegroup>
```

When you specify a predefined name attribute, omit the rest of the `jsxpalettegroup` tag's attributes—each built-in tab provides its own `display`, `picture`, and `tabOrder` attributes that you cannot change.

You can still specify an `order` attribute for your palette entries if you prefer to do so, as the preceding example does. This example adds its palette entries to the **Basic** tab. The `order` attribute's value of 3000 causes GoLive to add these palette entries after it adds the last built-in palette entry, which has an `order` value of 140. For a listing of all `taborder` and `orderid` values used by the built-in palette icons, see [“Objects Palette Entries” on page 196](#),

Adding Palette Entries to a Custom Tab

The following HTML example code creates the custom tab shown in [Figure 4.2](#).

```
<jsxpalettegroup name="XTND" display="Extension Sample"
                 tabOrder="3000" picture="paletteGroupIcon"
                 order=3000>
    // <jsxpalettentry> tags go here
</jsxpalettegroup>
```

As always, the name attribute defines the custom tab's JavaScript identifier, and the display attribute defines the tab's on-screen identifier; in this case, the display attribute defines this palette item's tool tip.

The tabOrder attribute defines a sort order for this tab among the other tabs that appear in the Objects palette.

This tag's order attribute is meaningful only when installing a custom palette entry amongst those built into GoLive itself. When a single extension places only its own entries on its own palette, as this simple example does, GoLive sorts the entries according to the order attribute of each entry's <jsxpalettentry> tag and it ignores the order attribute of the <jsxpalettegroup> tag that wraps those entries, and you can use any numeric values that produce your intended sort order. When you're installing your icon onto one of the built-in palettes, the order attribute defines a sort order for this palette's icons amongst all icons on the tab. The first two digits of this value specify the group of tabs amongst which this tab appears, while the second two digits define this tab's sort order within the group. The IDs used to sort the tabs and palette entries built into GoLive are listed in ["Objects Palette Entries" on page 196](#).

Basic Custom Boxes

When the user drags an icon from the Objects palette to a GoLive document, GoLive adds the tag's HTML code to the GoLive document and places a visual representation of the tag in the GoLive document window. This visual representation, called a **custom box**, can reflect the actual appearance of the custom element or it can be a placeholder graphic; for example, a placeholder graphic could be used to represent server-side content not available at the time the page is created. The custom box also provides JavaScript access to the attributes of the custom HTML element it represents.

When the user selects the custom box, GoLive

- Initializes the custom box as described in [Initializing the Custom Box](#).
- Displays the custom box as described in [Displaying the Custom Box](#).
- Activates the custom box's inspector window as described in [Inspecting the Custom Element](#).
- Provides handles that enable the user to resize the custom box, as described in [Resizing Custom Boxes](#). These handles do not appear when resizing behavior is disabled by the fixedWidth or fixedHeight attributes.

Initializing the Custom Box

GoLive initializes a custom box in response to any of the following events:

- The user drops an Objects palette icon onto a GoLive document window.
- GoLive reads a document containing a custom HTML element defined by the `jsxelement` and `jsxpalettentry` elements.
- The user switches to Layout View from another view in the document window.

To initialize the custom box, GoLive calls the `parseBox` function of the extension that created the element the box represents. Your implementation of this function must initialize the box in any way that is appropriate; for example, this function might set the box's height and width as specified by the `height` and `width` attributes of the custom HTML element.

GoLive passes to the `parseBox` function a [Box Object](#) that holds the HTML code associated with the custom element. This box object's `element` property provides JavaScript access to the custom element's attributes; thus, to initialize the `height` and `width` attributes of the custom box, your implementation of the `parseBox` function could use JavaScript code like that shown in the following example.

```
function parseBox(box) {  
    box.width = (box.element.width == undefined) ? 48 :  
                box.element.width;  
    box.height = (box.element.height == undefined) ? 48 :  
                 box.element.height;  
    box.url = (box.element.src == undefined) ? "none" :  
              box.element.src;  
    box.link = box.createLink(box.url);  
    box.oldWidth = box.width;  
    box.oldHeight = box.height;  
}
```

Displaying the Custom Box

To display the custom box, GoLive calls the extension's `drawBox` function, passing to it the box to draw and a Draw object. The [Draw Object](#) provides methods that draw lines, circles, rectangles, or images. Your implementation of the `drawBox` function calls the passed Draw object's methods to paint the visual representation of the custom box on the screen.

The Draw object is valid only during the execution of the function that receives it as an argument. Attempting to use the draw object outside this function generates a runtime error.

Here's a simple example of a drawBox function.

```
// define an img object to represent the placeholder graphic by name


function drawBox(box, draw) {
    // set width & height of graphic to values passed in by box
    placeholderGraphic.width = box.width;
    placeholderGraphic.height = box.height;
    // draw the placeholder graphic
    placeholderGraphic.draw(0,0);
}
```

This code sets the width and height of the placeholder graphic to the values specified by the custom box and then calls the draw function of the placeholderGraphic picture object to redraw the visual representation of the custom box at the size the user specified.

Here's a slightly more elaborate drawBox function. It paints a white background and frames it in black before drawing the custom placeholder graphic on top of this background.

```
// define an img object to represent the placeholder graphic by name


function drawBox(box, draw) {
    // draw a filled rect as background
    if (box.element.color != undefined)
        draw.setColor(box.element.color);
    else
        draw.setColor("white");
    draw.fillRect(0, 0, box.width, box.height);
    // frame the background in black
    draw.setColor("black");
    draw.frameRect(0, 0, box.width, box.height);

    // set width & height of graphic to values passed in by box
    placeholderGraphic.width = box.width;
    placeholderGraphic.height = box.height;
    // draw the placeholder graphic
    placeholderGraphic.draw(0,0);
}
```

The Draw object's setColor method specifies the color used for drawing. It accepts color values specified as any valid HTML color name, such as "red", "#FF0000", or "#F00". The method also accepts integer triplets specifying red, green, and blue values from 0 to 255. Therefore, the following three lines of JavaScript are equivalent—all set the color to red.

```
setColor ("#FF0000")
setColor ("red")
setColor (255, 0, 0)
```

The preceding `drawBox()` example always draws the same thing, however. An extension that adds multiple entries to the Objects palette would provide a `drawBox` function more like the following example, which uses a `switch` statement to call one of several `drawXx` custom drawing functions.

```
function drawBox(box, draw) {
    draw.textFont ("ApplicationFont");
    switch (box.element.type) {
        case "button":drawButton (box, draw); break;
        case "checkbox":drawCheck (box, draw); break;
        case "radiobutton":drawRadio (box, draw); break;
        case "edit":
            case "editarea":drawEdit (box, draw, false); break;
            case "buttonedit":drawEdit (box, draw, true); break;
        case "static":drawStatic (box, draw); break;
        case "color":drawColor (box, draw); break;
        case "urlgetter":drawGetter (box, draw); break;
        case "popup":drawPopup (box, draw); break;
    }
}
```

Each of these `drawXx` custom drawing functions uses the `draw` object in a manner similar to the following example of the `drawRadio` function.

```
function drawRadio (box, draw) {
    var w = box.width-1, h = box.height-1;
    draw.setColor ("silver");
    draw.fillRect (0, 0, w, h);
    var x = 5, y = (h-12)/2;
    draw.setColor ("white");
    draw.fillOval (x, y, 12, 12);
    draw.setColor ("black");
    draw.frameOval (x, y, 12, 12);
    draw.fillOval (x+3, y+3, 6, 6);
    var sh = draw.stringHeight(box.element.value);
    draw.moveTo (20, (h-sh)/2);
    draw.drawString (box.element.value);
}
```

For additional examples of the use of the `draw` object, see the custom `drawXx` drawing functions in the **Dialog Editor** sample code that the SDK provides.

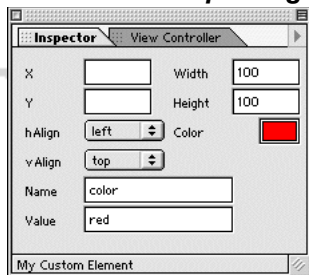
Inspecting the Custom Element

The **Inspector** window is always available in the Window menu. The **Inspector** window provides a user interface for getting and setting

- The properties of a custom element's JavaScript representation.
- The attributes of the HTML source the custom element added to a GoLive document.

When the user selects any modifiable page element in the GoLive document window, GoLive passes to the **Inspector** window a unique value that identifies the kind of element selected. The **Inspector** window's interface customizes itself accordingly, displaying the controls appropriate for manipulating that element's properties or attributes. GoLive provides inspectors for all built-in elements that the user can modify. To initialize the inspector window appropriately for a custom element, GoLive uses the custom element's `classid` attribute.

FIGURE 4.3 *Inspecting the Attributes of a Custom Element*



The `jsxinspector` tag specifies the group of `jsxcontrol` objects that populates the **Inspector** window when a custom element having a particular `classid` is selected.

```
<jsxinspector name="objectName" title="nameInWindowMenu"
              classid="yourUniqueID" width="anInteger" height="anInteger" >
    // jsxcontrol tags that provide inspector window controls go here
</jsxinspector>
```

Because the **Inspector** window is a palette window, defining a `jsxinspector` element is very similar to defining a `jsxpalette` element:

- Use the `name`, `title`, `width`, and `height` attributes just as you would use the same-named attributes of the `jsxpalette` tag. For more information, see [“Floating Palettes” on page 61](#).
- Use `jsxcontrol` tags to define the contents of your inspector dialog just as you would use them to define the contents of a `jsxpalette` window.

The `classid` attribute specifies the custom tag and custom palette entry that define the custom element this window inspects. Set the value of the `jsxinspector` tag's `classid` attribute to the same `classid` value used by the `jsxelement` and `jsxpaletteentry` tags that define the custom element this window inspects.

Here's an example of the use of the <jsxinspector> element. As you can see, it wraps around the numerous jsxcontrol elements that provide this inspector window's static text, edit fields, popup menus, and color field.

```
<jsxinspector name="myJSXInspector" title = "My Custom Element"
  classid="myClassID" width=215 height=200>
  <jsxcontrol type="static" posx=10 posy=14
    width=40 height=14 value="X">
  <jsxcontrol type="buttonedit" name="cX" posx=55 posy=10
    width=50 height=19>
  <jsxcontrol type="static" posx=10 posy=36 width=40 height=14
    value="Y">
  <jsxcontrol type="buttonedit" name="cY" posx=55 posy=32
    width=50 height=19>
  <jsxcontrol type="static" posx=110 posy=14 width=40 height=14
    value="Width" halign="right">
  <jsxcontrol type="buttonedit" name="cWidth" posx=155 posy=10
    width=50 height=19 halign="right">
  <jsxcontrol type="static" posx=110 posy=36 width=40 height=14
    value="Height" halign="right">
  <jsxcontrol type="buttonedit" name="cHeight" posx=155 posy=32
    width=50 height=19 halign="right">
  <jsxcontrol type="static" posx=10 posy=58 width=40 height=14
    value="hAlign">
  <jsxcontrol type="popup" name="cHalign" posx=55 posy=54
    width=60 height=19 value="left,center,right,scale">
  <jsxcontrol type="static" name="cColorLbl" posx=110 posy=58
    width=40 height=14 value="Color" halign="right">
  <jsxcontrol type="color" name="cColor" posx=175 posy=54
    width=30 height=19 halign="right">
  <jsxcontrol type="static" posx=10 posy=80 width=40 height=14
    value="vAlign">
  <jsxcontrol type="popup" name="cValign" posx=55 posy=76
    width=60 height=19 value="top,center,bottom,scale">
  <jsxcontrol type="static" posx=10 posy=102 width=40 height=14
    value="Name">
  <jsxcontrol type="buttonedit" name="cName" posx=55 posy=98
    width=100 height=19 halign="scale">
  <jsxcontrol type="static" posx=10 posy=124 width=40 height=14
    value="Value">
  <jsxcontrol type="buttonedit" name="cValue" posx=55 posy=120
    width=100 height=19 halign="scale">
</jsxinspector>
```

This code provides the **Inspector** window shown in [Figure 4.3 on page 78](#)

Initializing the Inspector Window

Before displaying the inspector window, GoLive calls the extension's `inspectBox` function, passing the box to inspect as its argument. Your implementation of this function initializes the controls in the **Inspector** window to reflect the current values of the properties of the JavaScript representation of the custom element. The following example illustrates a simple `inspectBox` function.

```
function inspectBox(box) {
    // init height and width fields per passed attrs
    box.inspector.cWidth.value = box.width;
    box.inspector.cHeight.value = box.height;
    // init other fields to defaults if no value supplied by box
    box.inspector.cHalign.value =
        (box.element.halign == undefined) ? "left" : box.element.halign;
    box.inspector.cValign.value =
        (box.element.valign == undefined) ? "top" : box.element.valign;
    box.inspector.cName.value =
        (box.element.name == undefined) ? "" : box.element.name;
    box.inspector.cValue.value =
        (box.element.value == undefined) ? "" : box.element.value;

    // enable & init color field only if it is selected
    var cEnabled = false;
    switch (box.element.type) {
        case "color": box.inspector.cColor.value = box.element.value;
                     cEnabled = true; break;
    }
    box.inspector.cColorLbl.enabled =
        box.inspector.cColor.enabled = cEnabled;
}
```

Most of this function body initializes the various controls the `jsxinspector` tag puts in the Inspector window. The `inspector` property of the box object GoLive passes to you provides access to the inspector window associated with this custom box; thus, each line of code that sets the value of a control in the inspector window is a variation on the following expression, which assigns a value to the control's value property:

```
box.inspector.controlName.value = newValue;
```

As you might guess, you can read a control object's value property to get the control's current value. Note that the object the value property holds, and the manner in which its value is set or retrieved, is a function of the control object's type.

The box object passed to this function by GoLive always has valid width and height properties, so the first two lines of code initialize the values of the `cWidth` and `cHeight` text-entry fields directly from these properties.

```
box.inspector.cWidth.value = box.width;
box.inspector.cHeight.value = box.height;
```

It's not necessarily good coding practice to assume that values passed to your code are valid, or even present. The next code fragment shows an initialization technique that tests for the presence of the `halign` property before using it. The `halign` and `valign` properties of the box

object specify the horizontal alignment and vertical alignment of the box with respect to its container; in this case, the container is the document window in which the box appears.

```
box.inspector.cHalign.value =  
    (box.element.halign == undefined) ? "left" : box.element.halign;
```

If this property is supplied, this code uses it; if not, the code sets the value of the `box.element.halign` property to a default value of `left` aligned. The next several lines of code use this technique to initialize most of the remaining controls in the **Inspector** window. Of interest is the final code fragment, which enables and initializes the **Inspector** window's color controls only when a color field is selected.

```
var cEnabled = false;  
switch (box.element.type) {  
    case "color":box.inspector.cColor.value = box.element.value;  
                cEnabled = true; break;  
}  
box.inspector.cColorLbl.enabled = box.inspector.cColor.enabled = cEnabled;
```

Although GoLive objects ignore bad values, it is recommended that your code always validate values before using them; when presented with bad input, your code can then more easily provide default values, display appropriate user feedback, or make a graceful exit.

The controlSignal Function

When the state of any control in the inspector window changes, GoLive calls the extension's `controlSignal` function, passing the changed control as its argument. Your implementation of this function must update the appearance of the custom box and the attributes of its markup elements.

The following code example depicts a `controlSignal` function that implements Inspector window functionality. The body of this function holds a `switch` statement that responds appropriately for the various controls GoLive passes to it.

```
function controlSignal(control)  
{  
    var box = control.parent.box;  
    switch (control.name) {  
        case "cX":box.element.posx = control.value; break;  
        case "cY":box.element.posy = control.value; break;  
        case "cWidth":box.width = box.element.width = control.value; break;  
        case "cHeight":box.height=box.element.height=control.value; break;  
        case "cHalign":box.element.halign = control.value; break;  
        case "cValign":box.element.valign = control.value; break;  
        case "cName":box.element.name = control.value; break;  
    }  
    box.refresh();  
}
```

The first line of code in the body of this example retrieves a reference to the custom box associated with the control that caused this method to be called.

```
var box = control.parent.box;
```

The parent property of the `jsxcontrol` object passed to the `controlSignal` function returns the **Inspector** window that displays the control. The **Inspector** window's `box` property provides access to the custom box that has the same `classid` as the inspector window. The `box` local variable holds this reference for use by subsequent statements in the body of the `controlSignal` function.

IMPORTANT: *The `box` property is valid only when the **Inspector** window is active; at all other times, the value of this property is `null`.*

Most of the rest of this function body is a `switch` statement that uses the `name` attribute of the passed control as its `case` expression. Each case in the `switch` statement assigns the current control value to the appropriate attribute of the custom element.

Once the element's attributes have been updated to reflect current control values, this function calls the `box` object's `refresh` method to redraw the custom box using the new values.

This particular example deals with only the controls defined by the **Inspector** window example shown earlier in this section; normally, your `controlSignal` function's `switch` statement provides a `case` expression for every control to which the extension must respond, including those that populate other custom dialogs or palettes your extension displays.

Resizing Custom Boxes

This section assumes that you understand the information presented in [“Basic Custom Boxes” on page 74](#). If you have not yet read this material, do so now, before reading this section.

When the user resizes a custom box, GoLive calls the extension's `boxResized` function, passing the `box` object and its new size as arguments. Your implementation of this function validates the new values and sets the attributes of the custom element accordingly, as the following example code does.

```
function boxResized(box, width, height) {
    if (width <= 500)
        box.element.width = width;
    if (height <= 500)
        box.element.height = height;
}
```

The validation you perform here is defined entirely by the specific needs of your extension; that is, bad values won't crash GoLive, but they may cause your extension to misbehave, so your validation code simply ensures that these values are suitable for your extension's use.

Built-In Undo Support

GoLive provides built-in undo support for dropping and resizing custom boxes. The user can undo or redo such operations by choosing the **Undo** or **Redo** item from the Edit menu.

For other operations involving your custom element, you'll need to implement undo support yourself, as described in [“Supporting the Undo and Redo Commands” on page 95](#).

Drawing Custom Controls

The visual representation of a custom control is a custom box. When a custom control needs to be redrawn, GoLive calls the `drawControl` global function, passing the control to draw and a [Draw Object](#) as its arguments. The Draw object provides methods that draw lines, circles, or rectangles.

Your implementation of the `drawControl` function calls the passed Draw object's methods to paint the visual representation of the custom control on the screen.

The Draw object is valid only during the execution of the function that receives it as an argument. Attempting to use the Draw object outside this function generates a runtime error. If the SDK detects an error in a drawing function, such as in the `drawControl` or `drawBox` function, it does not call this function any more. Without this feature, an error in a drawing function would cause GoLive to throw an endless stream of errors, because drawing functions are called whenever a JavaScript object must be redrawn.

Updating A Control's Appearance Immediately

The Draw object passed as an argument to the `drawControl` or `drawBox` function may not provide responsive enough drawing behavior for controls that provide user feedback through their appearance. To solve this problem, you can create a temporary Draw object that provides more responsive drawing behavior to the custom control.

The `beginDraw` method of a `jsxcontrol` object creates and returns a temporary Draw object that you can use to draw a control's new appearance immediately. When these drawing operations are complete, call the `endDraw` method to terminate the temporary Draw object.


The temporary Draw object is not valid after the `endDraw` method is called. Do not call the temporary draw object from outside the calls to the `beginDraw` and `endDraw` methods that create it and terminate it.

Only one temporary draw object can exist at any time. You cannot "nest" calls to the `beginDraw` method; that is, never call the `beginDraw` method more than once before calling the `endDraw` method.

Redefining Existing Tags

You can use the `<jsxelement>` tag to redefine any existing HTML tag; for example, you could define your own version of the `` tag or the `<MARQUEE>` tag. Redefining an existing tag effectively disables GoLive's built-in tag-handling code for that tag and turns it into a custom element that you manage like any other custom element you define; in other words, you must supply your own box as well as your own inspector for the redefined element.

When you replace a standard HTML tag with your own, your tag need not define its own palette icon or palette entry. You can install your new definition of the tag under the palette entry and icon GoLive provides for the standard tag.



When the user drags the icon to a GoLive document, the SDK adds your custom HTML to the page instead of the HTML specified by the standard tag you replaced. Instead of displaying the usual visual representation that GoLive displays for the replaced element, GoLive displays your custom box. The same thing happens when GoLive parses a file containing one or more occurrences of the redefined tag. Instead of creating the visual representation normally associated with the tag, GoLive creates custom boxes.

The ability to redefine tags as custom elements enables you to redefine a tag's inspector, or to define an inspector for an element that the user could not otherwise inspect. For example, the user cannot normally inspect the constituent elements of a GoLive component, which is a container that holds other HTML elements. When the same content is implemented as a custom tag that defines a custom page element and a custom box, a custom inspector can provide access to the properties of any of its constituent objects.

IMPORTANT: *Redefining a tag makes the standard inspector for that tag unavailable to all elements defined by that tag. For example, if you define your own version of the `` tag, `` elements cannot use the image inspector that GoLive provides—they must use a custom inspector you provide.*

Draft

5

Manipulating Document Objects

This chapter contains the following sections, which describe how your JavaScript code can manipulate GoLive document objects or their corresponding tags.

- [“The Markup Tree” on page 85](#)
- [“Selections” on page 88](#)
- [“Manipulating Elements Programmatically” on page 90](#)
- [“Supporting the Undo and Redo Commands” on page 95](#)

The Markup Tree

A **document** is a collection of objects in memory. A **file** is the disk-based representation of those objects. When GoLive reads an HTML file, it creates a document that holds a proprietary internal representation of the file’s data in memory and it leaves the file on disk unchanged.

When GoLive interprets a tag, it creates two sets of objects:

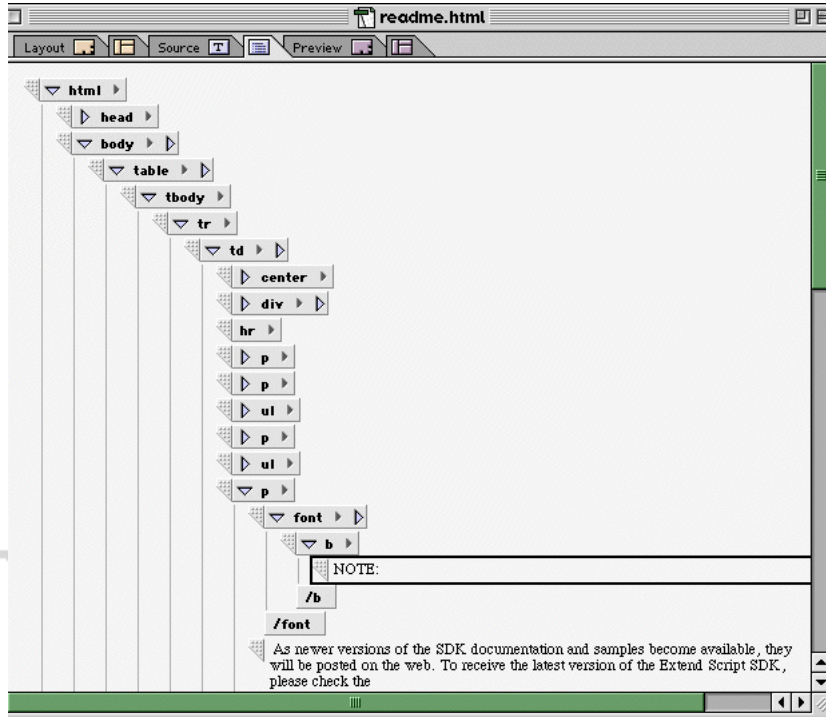
- An internal representation of the element that the tag and its attributes defines. This internal representation is not created in JavaScript and JavaScript callers cannot access it directly.
- A set of JavaScript objects that provide access to the internal representation of the element. These objects are structured as a binary tree, known as the **markup tree**. Individual objects in the tree are called **markup elements**. The structure of the markup tree resembles that of the HTML elements that define it.

Each markup element represents an HTML element definition in the source HTML. The properties of the markup element implement the attributes of its associated HTML element. For example, if your HTML element defines the `myAttr` attribute, the markup element it generates has a `myAttr` property.

GoLive provides two structured views of data associated with the active document:

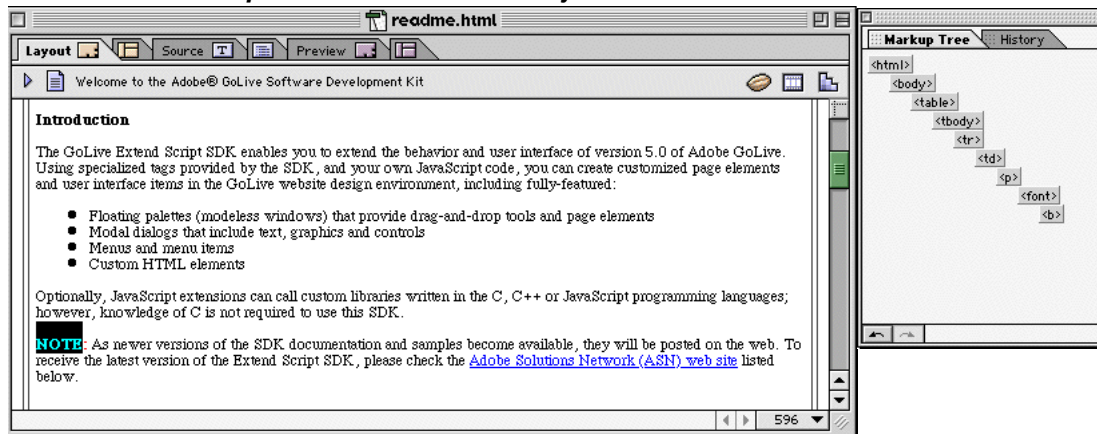
- The **HTML outline view**, shown in [Figure 5.1](#), provides a graphical depiction of the element hierarchy that the active document’s HTML source defines. You can use this view to help you envision the structure of the markup tree that this source generates; in particular, you can look at this view to see how individual elements fit into the entire tree. You can edit HTML source directly in this view, but you cannot edit the markup tree directly in this view.

FIGURE 5.1 HTML Outline view



- The **Markup Tree** window, shown in [Figure 5.2](#), depicts the branch of the markup tree that leads from the currently-selected object to the root node of the tree. This view is useful for determining exactly which markup elements provide access to the currently-selected element. You can also click objects in this view to set the selection in the document window.

FIGURE 5.2 Markup Tree window shows objects that contain current selection



The root of the tree is an object that corresponds to the `<GoLiveMarkup>` tag reserved for use by GoLive. This root object defines one `html` element, which defines a `body` element and one or more optional `head` elements. The `body` element can contain additional `GoLiveMarkup`

elements, each of them being the root of a tree of objects that represents the content of a single page in the site. Each optional head element can provide text, scripts, or both.

JavaScript Access to the Markup Tree

The JavaScript **document object model** (DOM) provides access to the markup tree from JavaScript. This section introduces the most significant objects, properties, and methods used to work with the markup tree. To familiarize yourself with these objects, you can use the **JavaScript Shell** window to evaluate some of the JavaScript expressions this section describes.

Almost every SDK object provides a `name` property that you can use to retrieve it. The objects that represent a document and its markup tree are notable exceptions to this rule. As this section describes how to access these objects, it will also note how they are identified in the JavaScript namespace. For your future reference, [Table 5.1](#) summarizes this information.

TABLE 5.1 *Identifiers for Document and markup objects*

Type of Object	Property used for access by name
JSXDocument	<code>title</code>
JSXMarkup	<code>tagName</code>

The global `document` property returns a [Document Object](#) that represents the currently-active GoLive document. For example, if you open the `readme.html` file in GoLive and enter `document` into the JavaScript Shell window, GoLive returns `[JSXDocument readme.html]` as the result. The `JSXDocument` identifier means this object is an Extend Script [Document Object](#), and that `readme.html` is the value of this document object's `title` property.

The `element` property of the document object holds the markup tree as a [Markup Object](#), which provides properties and methods you can use to

- navigate the markup tree.
- manipulate markup elements and associated HTML sources programmatically.
- retrieve subelements.
- retrieve its parent element.

Of course, the root of the tree does not have a parent element.

Evaluating `document.element` returns `[JSXMarkup <GoLiveMarkup>]` as the result. The `JSXMarkup` identifier indicates that this object is a markup object, and `<GoLiveMarkup>` is its `tagName` property. This particular `JSXMarkup` object is special because its `tagName` property identifies it as the `JSXMarkup` object that is the root of the markup tree. Only one `JSXMarkup` element per page has `<GoLiveMarkup>` as the value of its `tagName` property.

The rest of the markup tree consists of JSXMarkup objects that are the subelements of the <GoLiveMarkup> object. You can retrieve a subelement in the following ways:

- The subelements property of the Markup object presents the subelements as an array that can be accessed by name or by number.
- The getSubElement method of the Markup object retrieves the *n*th occurrence of a subelement having a specified tagName property. This method counts elements depth-first.

For details regarding all of the properties and methods this object provides, see [“Markup Object” on page 152](#).

The JSXMarkup object also furnishes methods that provide access to the HTML code that defines the element it represents. These methods can separate an element’s opening and closing tags from the HTML they surround. An element’s opening tag and closing tag are collectively referred to as the element’s **outer HTML**, while everything appearing between these tags is known as the element’s **inner HTML**. In the following example, the element’s outer HTML consists of the <H1> start tag and the </H1> end tag, while its inner HTML consists of the This is the inner HTML text.

```
<H1> // outer HTML
  This is the inner HTML
</H1> // outer HTML
```

The concept of inner and outer HTML applies only to binary tags, which are those used in pairs consisting of an opening tag and a closing tag. For example, a unary tag such as the tag does not require the use of a closing tag; thus, the getInnerHTML method returns the empty string for this element.

Selections

The document.selection property holds a [Selection Object](#) that provides JavaScript access to the current user selection in the active document.

The user can create different kinds of selections; for example the user can select part of an element, an entire element, or multiple elements. The value of the selection object’s kind property provides a hint about what the user selected:

point	No selection. The selection reflects the cursor position.
part	A portion of the current markup element is selected, such as when only part of a text element is selected.
full	The entire markup element has been selected, such as when the user clicks an image box.
complex	More than one markup element is selected or partially selected.
outside	The selection is outside of the current markup element. This should not happen.

Three additional properties provide access to the HTML source associated with the selected elements:

<code>start</code>	<i>Number</i>	Offset from the outer HTML representation of the first selected tag to the first character of HTML code in the selection.
<code>length</code>	<i>Number</i>	Number of ASCII characters that define the selection in its associated HTML.
<code>text</code>	<i>String</i>	The HTML source code associated with the selection.

The selection object's `element` property holds the first markup element that is part of the current selection.

Retrieving the Current Selection

The following example demonstrates how to use the properties of the `selection` object to retrieve the current user selection in the active document. If the user selects the word `is` in a text element that holds the phrase `This is a fine product` as its content, the properties of the selection object would provide the following information:

Property	Value	Description
<code>document.selection.element</code>	"Text"	Markup object that holds selection. This element is a Text element.
<code>document.selection.type</code>	"part"	Only part of the element is selected.
<code>document.selection.start</code>	5	Selection begins at offset position 5.
<code>document.selection.length</code>	2	Selection is two characters in length.
<code>document.selection.text</code>	"is"	The selected characters.

When the selection is not text, the `document.selection.text` property holds the HTML source that defines the selection; for example, clicking an image in the active GoLive document results in the following selection data:

Property	Value	Description
<code>document.selection.element</code>	"Img"	Markup object that holds selection. This element is an <code>Img</code> element created by the <code></code> tag and attributes in the <code>document.selection.text</code> property.
<code>document.selection.type</code>	"full"	The entire element is selected.
<code>document.selection.start</code>	0	Selection is not offset from beginning of element.

Property	Value	Description
<code>document.selection.length</code>	24	The source text that defines the selected markup element is 24 characters long. The <code>document.selection.text</code> property provides access to this text.
<code>document.selection.text</code>	""	The selected object's HTML definition.

Setting the Current Selection

To select a markup element, assign it to the `element` property of the `selection` object. Assigning to this property anything other than a markup element returns an undefined result and leaves the current selection unchanged.

The following example selects the third image in the active document.

```
document.selection.element =
    document.element.getSubElement ("img", 2);
```

NOTE: The SDK does not support programmatic selection of a partial element; for example, Javascript code cannot select the third word in a headline, but the end-user can make a partial selection such as this one.

To make a window frontmost, assign its document object to the global `document` property.

Manipulating Elements Programmatically

Changing an element of the markup tree does not change that element's HTML representation in the document that generated the markup tree, nor does it change the element's visual representation in the GoLive document window. Generally, whenever you change one of these things, you must update the others. This section describes how to

- add, delete, or change HTML elements in a GoLive document
- generate a new markup tree that reflects the current HTML
- update the document's visual representation on screen to display the new markup.

To change the HTML source code associated with an element, use the `getInnerHTML`, `getOuterHTML`, `setInnerHTML`, and `setOuterHTML` methods as necessary to manipulate the tags that define the element in the source document. The new HTML code you insert in the source document does not have to resemble the code it replaces—it can be completely different if you so choose.

The following example defines an extension-specific `makeBold` function that sets the current text selection to boldface by enclosing it between `` and `` tags; to do so, it replaces the selected element's inner HTML with HTML that includes these additional tags.

```

sel = document.selection;
function makeBold (sel) {
    // exit if selection is null
    if (sel == null) return;
    // work with full or partial selections only
    if ((sel.type == "full") or (sel.type == "part"))
    {
        var selText = sel.text;
        var htmlText = sel.element.getInnerHTML();
        htmlText = htmlText.substr (0, sel.start)
            + "<b>" + selText + "</b>"
            + htmlText.substr (sel.start + sel.length);
        sel.element.setInnerHTML (htmlText);
        document.reparse();
    }
    // display appropriate alerts for invalid selections
    if (sel.type == "point")
        alert("You haven't made a selection.");
    if (sel.type == "complex")
        alert("You've selected more than one element.");
    if (sel.type == "outside")
        alert("Sorry, selection is not valid.");
}

```

The first line of code in this example retrieves the [Selection Object](#) that provides JavaScript access to the current user selection in the currently-active GoLive document.

```
sel = document.selection;
```

The next line of code passes this object to the makeBold example function.

```
function makeBold (sel) {
```

Before attempting to work with the selection at all, the makeBold function performs several tests to ensure the validity of selection passed to it. If passed a null object, this function simply exits without taking any further action.

```
    if (sel == null) return;
```

The next line of code tests the type property of the selection object to determine whether the selection is one that this function can manipulate successfully. In this case, full or partial selections are valid. Any other kind of selection results in the display of a user alert.

```

    if ((sel.type == "full") or (sel.type == "part"))
    {
        // code to manipulate valid selections goes here
    }

    // display appropriate alerts for invalid selections
    if (sel.type == "point")
        alert("You haven't made a selection.");
    if (sel.type == "complex")
        alert("You've selected more than one element.");
    if (sel.type == "outside")
        alert("Sorry, selection is not valid.");
}

```

The next block of code places the currently-selected text between `` and `` tags by constructing a new text string that includes the original text and the new tags. The `text` property of the selection object holds the currently-selected text; thus, the first line of the body of this function uses this property to extract the text selection and store it in the `selText` local variable.

```
var selText = sel.text;
```

Next, the function retrieves the HTML to replace.

```
var htmlText = sel.element.innerHTML();
```

The selection object's `element` property holds the first element in the selection. That element's `getInnerHTML` method returns that portion of the element's HTML which appears between its opening and closing tags (or, in the case of a unary tag, the text that appears between its opening and closing angle brackets). This line of code stores the retrieved inner HTML in the `htmlText` local variable.

Having retrieved the original text that appears on the screen, as well as the inner HTML that defines it, this function can now use them to construct a string to insert in the document as the new inner HTML which boldfaces the current text selection. To construct this string, the function uses the `+` operator and the `substr` method that all JavaScript string objects provide. The `substr` method dissects the original HTML text into pieces that the `+` operator combines with the `""` and `""` string literals to create a new text string which, when interpreted as HTML, renders the originally-selected text in boldface.

The following line of code does this work. The next several paragraphs examine this code in detail.

```
htmlText = htmlText.substr (0, sel.start)
           + "<b>" + selText + "</b>"
           + htmlText.substr (sel.start + sel.length);
```

Because the JavaScript expression to the right of the `=` operator is evaluated before it is assigned to the `htmlText` variable that appears to the left of the `=` operator, the right-hand side of this statement operates on the contents of the `htmlText` variable to construct the new string before the left-hand side reuses the `htmlText` variable to hold the newly-constructed string.


The syntax of the `substr` method looks like this:

```
returnedText string.substr (start, length);
```

The `substr` method returns a portion of the *string* object's text. The *returnedText* holds *length* number of characters of the original *string* object's *value* property, beginning at the *start* index position. The first character in a JavaScript string is at position 0.

The `start` property of the selection object specifies the position of the first selected character that its `text` property provides. Because you are replacing the entire element, not just the characters to be boldfaced, the constructed string must include those characters that are not part of the current selection. Thus, the constructed string begins with the `htmlText.substr (0, sel.start)` expression.

Because indexes into JavaScript strings are zero-based, the 0 argument specifies that the returned substring begins with the first character in the `htmlText` string. Similarly, passing the `sel.start` expression as the `length` argument causes the `substr` method to include in the



returnedText string all of the characters up to (but not including) the first selected character. For example, if you have a string that is 15 characters in length, the first character in the string is at index position 0 and the tenth character in that string is at index position 9; thus, if the tenth character is selected, the value of `sel.start` is 9. For this example string, `substr (0, sel.start)` would evaluate to a string that includes 9 characters, beginning at position 0, that includes the characters in positions 0 through 8. Thus, using `sel.start` as the `length` argument to the `substr` method conveniently excludes the first selected character from the substring that begins the newly-constructed HTML code.

The next portion of this expression uses the `+` operator to append the `""` string to the HTML being constructed. The `+` operator is then used again to add the `selText` variable to the substring being constructed; because `selText` holds the currently-selected text, adding it to the substring has the effect of adding this text immediately following the `` tag. A string literal defining the `""` closing tag then follows to end the boldfacing of text in the string; again, the `+` operator adds this tag to the string being constructed.

To finish the construction of the new HTML, this code appends the result of the `htmlText.substr (sel.start + sel.length)` statement to the string being constructed. Note that `(sel.start + sel.length)` evaluates to a single argument to the `substr` method; when you supply only one argument to the `substr` method, that argument is assumed to specify the index position of the first character in a substring that includes all characters from that position through the end of the string. Thus, the `(sel.start + sel.length)` expression evaluates to the index position of the first character following the current selection, and the `htmlText.substr (sel.start + sel.length)` statement returns all of the characters from this position through the end of the string. Thus, using the `+` operator to append this expression to the string being constructed appends the rest of the original text string to the new HTML being constructed. At this point, evaluation of the expression to the right of the `=` sign is complete, so the newly-constructed string is assigned to the `htmlText` variable.

To replace the selected element's HTML representation with the newly-constructed HTML code, pass the `htmlText` variable as the argument to the element's `setInnerHTML` method, as the following example does.

```
sel.element.setInnerHTML (htmlText)
```

You must call the document's `reparse` method immediately after calling either of the `setInnerHTML` or `setOuterHTML` methods. The HTML that these methods place in the source document is not represented in the markup tree or in the active GoLive document until you call the active document's `reparse` method.

```
document.reparse();
```

The `reparse` method generates a new markup tree of objects representing the current set of HTML elements the document defines and updates the document's visual representation in the GoLive environment.

Whenever GoLive creates a new custom box, it calls your extension's `parseBox` method once for that box. GoLive creates a new custom box whenever it reads a tag. A variety of situations can cause this to occur:

- GoLive reads a file containing markup tags.
- the user drops an Objects palette entry onto a GoLive document.
- the user pastes an element into a GoLive document.
- the user changes the size or location of an existing box in a GoLive document.
- GoLive generates a new markup tree, perhaps due to user interaction with the layout window, or because a script called the `reparse` method.

When you generate a new markup tree, all JavaScript objects referring to the previous tree become invalid; thus, your implementation of the `parseBox` method must update your extension's saved references to JavaScript objects.

NOTE: To ensure that GoLive can parse all elements successfully, calls to the `parseBox` method are executed after a short delay.

With due caution, you can use the `setInnerHTML` and `setOuterHTML` methods to replace virtually any element in the source document. However, you should exercise great care when using these methods. `setInnerHTML()` replaces all text within the markup element. Calling `setInnerHTML("hello")` on an `` would change the text to `<hello>`, and a tag [whatever] would become [hello]. For binary tags, the text between the tag and its end tag is changed, not the contents of the tag itself. The tag `<object>sometext</object>` would thus become `<object>hello</object>`. The `setOuterHTML()` method replaces the entire tag including the surrounding characters and an end tag with the given text. Calling `setOuterHTML("hello")` on `` would become `hello`, as the tag `<object>sometext</object>` would become

Using `setOuterHTML()` or `setInnerHTML()` on tag elements is very dangerous and should be avoided, since only the source buffer is updated. The entire markup tree may become invalid when the content of a tag element is replaced; to avoid crashes, the document must be reparsed immediately after any `setInnerHTML` or `setOuterHTML` method call, especially if that call modifies a tag element.

IMPORTANT: *To continue working with the markup tree after using `setOuterHTML()` or `setInnerHTML()` on a tag element without reparsing is especially dangerous—it leads to undefined results that are very likely to crash GoLive.*

HTML tags inside a text element as in the above example are not visible in the markup tree until the document is reparsed. Calling the method on non-HTML tags like text or comment items does not invalidate the markup tree, but the changes are not visible until the document has been reparsed. To reparse the current document, call the `document.reparse()` method.

Supporting the Undo and Redo Commands

GoLive provides built-in undo support for dropping and resizing custom boxes. The user can undo or redo such operations by choosing the **Undo** or **Redo** item from the Edit menu. For other operations involving your custom element, you must implement undo support yourself. The user cannot choose the **Undo** or **Redo** command to reverse the effects of other interactions with your extension unless you implement code that provides this functionality, as described in this section.

Creating the Undo Object

Each operation that can be undone must provide an [Undo Object](#) that holds the data used to perform the operation, to undo its effects, and to redo its effects. Normally, you must create this object from within the body of the function that performs the operation to be undone. This function might be one of your own extension-specific functions or it might be your implementation of an event handler function that GoLive calls. For example, your extension's `controlSignal` method might create an undo object that GoLive uses to undo changes to a control's current setting.

The `boxResized` function is an exception to this rule. Because GoLive provides built-in undo support for dropping and resizing custom boxes, your `boxResized` method need not create its own undo object.

NOTE: If your `boxResized` function creates its own undo object, it may interfere with the built-in undo action GoLive already provides for the resizing operation. If this happens, your script terminates with the runtime error message "There is already an open undo action."

For other operations, you'll create an undo object by calling the `createUndo` method of the [Document Object](#) from within the body of the function that performs the operation to be undone. As the following example does, store the `createUndo` method's return result in a local variable.

```
// this extension calls the undoable fn from the controlSignal fn
function controlSignal(control) {
    switch (control.parent.name){
        {case "URL": setURL(control);break;
        // other cases here...
        }
    }
}
// this extension initializes the current url value when the box is parsed
function parseBox (box) {
    box.url = "none";
}
// save old & new URLs into undo object & submit undo object to GoLive
function setURL (box,url) {
    var undo = document.createUndo ("Set URL");
    undo.kind = "Link";
    undo.box = box.name;
    undo.oldURL = box.url;
    undo.newURL = url;
    undo.submit(); // we set the URL on Do
}
```

GoLive uses the `createUndo` method's argument to customize the display of **Undo** and **Redo** items in the **Edit** menu; in this example, the `Set URL` argument causes GoLive to display **Undo Set URL** and **Redo Set URL** menu items as necessary.

Initializing the Undo Object

The `createUndo` method creates an “empty” undo object. You must store in this object any properties your function needs to perform an operation, undo its effects, or redo its effects. You can add properties to the undo object by simply assigning them to it, as follows.

```
undo.kind = "Link";
undo.oldURL = box.oldURL;
undo.newURL = url;
```

The exact set of properties your undo object holds is an extension-specific implementation detail. In this particular example, the undo object holds old and new URLs the user has assigned to the custom element, but another function's undo object could hold color values, custom tags, or other kinds of data.

You'll probably find it useful to add `kind` and `name` properties to your undo object:

- If your extension implements more than one command that can be undone, you'll need to implement a property that you can use to determine which command is to be undone; in this example, the `kind` property serves that purpose.


```
undo.kind = "Link";
```
- You may also need to store something you can use to retrieve the undo object later, such as a unique name property.


```
undo.boxName = box.name;
```


When you've finished adding properties to the undo object, pass it back to GoLive by calling the undo object's `submit` method, as follows.

```
undo.submit();
```

As soon as the undo object is submitted, GoLive makes it available to the user in the **History** palette. If necessary, you can still add properties to an undo object even after it has been submitted; however, this is recommended only for properties that cannot be stored within the body of the function that creates the undo object.

Implementing the `undoSignal` Function

When the user chooses the **Undo** item or when you call the `submit` method, GoLive passes the undo object to your extension's `undoSignal` method, along with an action code indicating whether your method should do the operation for the first time, undo the operation, or redo the operation. The values of this action code and the states they represent are:

0	Do. Undo object's <code>submit</code> method was called. Do the operation for the first time.
1	Undo. User issued the Undo command. Undo the operation.
2	Redo. User issued the Redo command. Reverse the effects of the Undo operation.

Your implementation of the `undoSignal` method utilizes the action code and previously-stored undo object properties to respond appropriately; for example, your `undoSignal` function may need to change the appearance of a custom box and manipulate the document objects and markup elements it represents.

The `kind` property created and saved previously is used as the case expression in a `switch` statement that customizes the actions of the `undoSignal` function according to the particular undo object GoLive passes to it.

```
function undoSignal (undo,action) {
    switch (undo.kind) {
        case "Link":undoLink (undo, action); break;
        // assume we've created these additional undo actions
        case "TextColor":undoTextColor (undo, action); break;
        case "What":undoSomethingElse (undo, action); break;
    }
}
```

Because each case in an `undoSignal` function must handle three cases of its own (Do, Undo, and Redo), you may improve the readability of your code by creating your own functions that the cases of the `undoSignal` function can call to perform tasks. The next code example illustrates such an approach. This `undoLink` function provides the code to undo a change in the `url` property of a custom box.

```
function undoLink (undo, action) {
    var box = boxes[undo.boxName]; // get the real box behind the name
    if (action == 1) { // undo
        box.url = undo.oldURL;
    } else { // do or redo
        box.url = undo.newURL;
    }
}
```

The first line of the body of this function retrieves the custom box that holds the `url` property this function changes. To do so, it extracts the previously-saved `boxName` property from the `undo` object passed to this function and uses this name to retrieve the box from the `boxes` array GoLive maintains in the JavaScript global namespace. It then stores the retrieved box object in the `box` local variable.

```
var box = boxes[undo.boxName]; // get the real box behind the name
```

The function's next action is conditioned on the value of the `action` code GoLive passes to it. An action code value of 1 indicates that the user issued the Undo command, which means this function must restore the box's `url` property to its previous value. When the `setURL` function created the `undo` object, it saved this value as the `oldURL` property. Thus, restoring the `url` property of the box is straightforward—we simply assign the value of the `undo` object's `oldURL` property to the box object's `url` property, as follows.

```
if (action == 1) { // undo
    box.url = undo.oldURL;
} else { // do or redo
    box.url = undo.newURL;
}
```

In this simple example, the `url` property can hold only two possible values, so this function can treat the **Do** and **Redo** action codes exactly the same; of course, your functions can handle the **Do** and **Redo** action codes separately if necessary.

IMPORTANT: *Do not call the `reparse`, `reformat`, `setInnerHTML`, or `setOuterHTML` methods of the [Document Object](#) from within an undo action.*

Accessing the Document History

The `Document.history` property returns a [History Object](#) you can use to inspect or manipulate the active document's undo/redo history. Additionally, you can set the history list's `index` property to cause GoLive to execute any undo or redo actions necessary to restore the document to the state that the specified index represents.

6

Files

The [File Object](#) enables JavaScript code to read, write, create, delete, move, copy, and rename files on local and remote file systems. Additionally, this object can return a list of the files residing in a specified folder.

Creating a File Object

You can acquire a file object from the Application object, or you can create one yourself.

- Properties of the Application object provide File objects that reference commonly-used directories such as the one that holds the GoLive application; for details, see the [Built-in Access to Commonly-Used Folders](#) section.
- To access other directories or files, you must create your own File objects explicitly, as described in the [Creating A File Object Explicitly](#) section.

Built-in Access to Commonly-Used Folders

The following properties of the [Application Object](#) provide File objects which refer to commonly-used folders such as the application folder, system folder, and trash folder.

folder	<i>File</i>	A <i>File</i> object referring to the folder that holds the GoLive application. Read-only.
currentFolder	<i>File</i>	Returns a <i>File</i> object referring to the current folder. May be assigned a path name or a <i>File</i> object describing a folder.
settingsFolder	<i>File</i>	A <i>File</i> object referring to the folder that holds GoLive extension modules and other settings. Read-only.
systemFolder	<i>File</i>	A <i>File</i> object referring to the folder that holds the operating system. Usually, this is the C:\WINDOWS or C:\WINNT folder on Windows platforms; on Mac OS platforms, this folder can have any name and it can reside on any local disk or network volume.
tempFolder	<i>File</i>	A <i>File</i> object referring to the folder GoLive uses to store temporary files. Read-only.
trashFolder	<i>File</i>	A <i>File</i> object referring to the system Trash folder. On Windows platforms, this object refers to the local \RECYCLED directory. On Mac OS platforms, this object refers to the <i>startupDisk</i> :Trash folder. Read-only.

Creating A File Object Explicitly

To create a file object explicitly, you can call the `JSXFile` global function or you can use the new operator on the `JSXFile` constructor function, as in the following examples.

```
// global function
var theFileObject = JSXFile(fileOrFolderPathName)

// constructor
var theFileObject = new JSXFile(fileOrFolderPathName)
```

Both functions accept as their sole argument the name of the file or folder that the newly-created file object is to represent. This argument may be specified as a partial path name, a full path name, or a local URL. If you do not specify this argument as a URL, you must ensure that your calls to the `JSXFile` constructor use the separator character and file-naming conventions expected of that platform. At the minimum, this involves using a colon character (:) to separate directories in Mac OS filenames, and using a backslash character (\) to separate directories in Windows pathnames. If you specify no argument at all, the `JSXFile` constructor creates a File object that refers to the current directory.

For example, if the current directory on a Windows machine is `D:\GoLive`, each of line of code in the following example creates a File object that represents the `D:\GoLive\main.html` file.

```
file = new JSXFile ("D:\\GoLive\\main.html");
file = JSXFile ("main.html");
file = new JSXFile ("file:///d:/GoLive/main.html");
```

Testing For the Presence of a File or Folder

A newly-constructed `JSXFile` object holds only the full pathname of the file or folder it represents. This file or folder does not necessarily exist—you might created a file object in order to call methods that create the file or folder programmatically. To determine whether the file object's file or folder exists, you can test the value of the file object's `exists` property. The value of the `exists` property is `true` when the file or folder represented by the file object exists.

Determining What the File Object Represents

To determine whether the file object represents a file or a folder, you can test the value of the File object's `isFolder` property. The value of the `isFolder` property is `true` when the `JSXFile` object references a folder; when the value of this property is `false`, the `JSXFile` object represents a file. The value of this property, too, does not indicate whether the file or folder exists.

Creating A Folder Programmatically

To create a folder programmatically, call the `createFolder` method of the `File` object that represents that folder. This method creates a folder having the name and location specified by the path name that the `File` object encapsulates.

Retrieving Files Programmatically

To retrieve files from the folder a `JSXFile` object represents, call the `JSXFile` object's `getFiles` method. If you call the `getFiles` method of a `JSXFile` object that does not represent a folder, the method returns `null`.

```
Array fileObject.getFiles([String mask, String type])
```

This method returns an array of `JSXFile` objects. Each file object in the array corresponds to a file having a name that matched the search criteria passed as the arguments to this method.

The *mask* parameter specifies a filename-search string that can contain wildcard characters such as questions marks and asterisks. These wildcard characters are interpreted as on Windows systems:

- An asterisk (*) represents zero or more occurrences of the item that precedes it in the search string.

NOTE: To retrieve all files in a folder, you need not specify any search mask; by default, the `getFiles` method uses a search mask that consists of the * character only.

- A question mark (?) represents zero or one occurrences of the item preceding it in the search string

The optional *type* parameter specifies a Mac OS file type as a four-byte string, such as `TEXT` or `JPEG`. Windows systems ignore this parameter.

Retrieving A File's Location

A `JSXFile` object's `parent` property holds a `JSXFile` object that represents the folder in which the file resides. The value of this property is `null` for a file that resides in topmost folder of the file system.

Moving Files and Folders

To move a file or a directory, call the `move` method of the `JSXFile` object that represents it. You can specify the new location for the file as a pathname or as a local URL.

You cannot use the `move` method to transfer files to remote volumes; to upload files to a remote server volume, use the `put` method of the `JSXFile` object that represents the file or directory to upload. For more information, see [“Uploading Files To Remote Volumes” on page 102](#).

Copying Files and Folders

To copy a file or a directory, call the `copy` method of the `JSXFile` object that represents it. You can specify the new location for the file as a pathname or as a local URL.

You cannot use the `copy` method to transfer files to remote volumes; to upload files to a remote server volume, use the `put` method of the `JSXFile` object that represents the file or directory to upload. For more information, see [“Uploading Files To Remote Volumes” on page 102](#).

Uploading Files To Remote Volumes

The `get` and `put` methods of the `JSXFile` object provide simple HTTP download and upload services. With the help of these two methods, you can download a file, edit it, and then upload it again. These methods require that the **Network** module be active; for information on activating GoLive modules, see [“Enabling the Extend Script Module” on page 26](#).

The `get` method retrieves a specified file from an HTTP server and stores it on disk under the name stored inside the `JSXFile` object, while the `put` method uploads a file to an HTTP server. The `put` method requires that the destination server be able to fulfill HTTP PUT requests.

Each of these methods accepts a remote URL as its argument. An optional second parameter specifies the file's MIME type.

When either of these methods return, the `JSXFile` object's `lastError` property holds a string that is the HTTP status text returned by the server, such as "200 OK", "201 Created" or "404 Not found".

You can set the `lastError` property to any string value; to ensure that this property contains the latest error text, assign an empty string as its value before attempting a file I/O operation.

This chapter describes the following additional programming topics:

- [“Working With Pictures” on page 103](#)
- [“Timed Tasks” on page 104](#)
- [“Persistent Data” on page 104](#)
- [“Progress Bars” on page 105](#)
- [“Localization” on page 106](#)

Working With Pictures

Currently, the SDK supports pictures only as `Picture` objects returned by the `createPicture` global function.

Creating Pictures

To create a new `Picture` object, call the `createPicture` global function, as in the following example.

```
myPicture = createPicture (url) // create Picture dynamically
```

You cannot create a valid `Picture` object by using the `new` operator on the `Picture` constructor, as in the following example.

```
myPicture = new Picture (url) // not supported
```

Deleting Pictures

To delete a picture, do not use the `delete` operator; instead, call the `disposePicture` (`picture`) global function, as in the following example.

```
myPicture = createPicture (url)
result = disposePicture (myPicture)
```

Releasing Saved JavaScript References

Before unloading your extension, GoLive calls its `terminateModule` function. Your implementation of this function must release all of your extension’s saved JavaScript references by setting to null the values of all global variables your extension creates.

For debugging purposes, there several ways you can force GoLive to unload your extension's scripts. See [Chapter 8, "Debugging"](#) for information on

- Using the **Reload Scripts** and **Unload CAPIs** items that the JavaScript Output palette's menu provides.
- Using the `unload` function of the `Module` object to unload an external library.

Timed Tasks

A portion of Javascript code can be scheduled for a later execution. The global function `startTimer()` accepts a scriptlet and a timeout. The scriptlet is stored internally and executed as soon as the timeout has elapsed. Optionally, the scriptlet can be scheduled for repeated execution so it is executed f.ex. every second.

The following code would print a counter in the Javascript Output window every second:

```
counter = 0; myTimer = startTimer ("writeln (++counter)", 1000, true);
```

To stop this code, use the following statement:

```
stopTimer (myTimer);
```

Persistent Data

Extension modules can create their own preference data which is maintained for all extension modules and is persistent across multiple runs of GoLive. This feature enables all extensions to share a common set of preferences and to store persistent data. The preferences are accessed via the global object `prefs`. The properties of this object are all stored as preferences, so creating a new preference value is as easy as writing to a `prefs` property. If, for example, one module executed the code

```
prefs.myModule = "Version 1.0";
```

All other modules can check for the presence of this module with the code

```
if (prefs.myModule == "Version 1.0")...
```

The preference value is saved to disk along with all other preference data and is available to the scripts when GoLive is started for the next time. Javascript objects cannot be saved.

The GoLive Extend Script SDK has limited read-only access to the GoLive preferences. These preferences may be accessed via the `app.prefs` object.

Progress Bars

To display user feedback during lengthy operations, your extension can use the progress bars and busy indicators built into GoLive.

FIGURE 7.1 *Progress Bar*

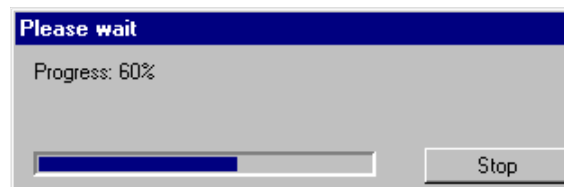


FIGURE 7.2 *Busy Bar*



Three methods of the [Application Object](#) provide these services:

- *startProgress()* initializes and displays the bar.
- *setProgress()* updates the status bar or busy bar display.
- *stopProgress()* hides the progress bar.

Application.startProgress() takes four parameters. The first is the window title. The second is the optional initial status message. The third parameter is a boolean value indicating whether a progress bar or a busy bar should be displayed. If the parameter is *true*, a busy bar is used. The last, optional parameter takes the number of seconds which should pass before the window pops up. Setting this value to, say, two seconds does not display the window when an operation takes less than two seconds.

To display the progress of an operation and to scan the status of the *Stop* button, call the method *Application.setProgress()*. When displaying a progress bar, the first parameter is the progress value, which must be a value between 0 and 1. The second, optional parameter is a new status message. When used with a busy bar, there is no need to supply any parameters unless you want a different status message to be displayed. In this case, the progress value is ignored. This method also rotates the barbershop indicator about once a second. When the user clicked the *Stop* button, this method returns *false*. You should call this method frequently during your lengthy operation to show the user that the program still is alive.

Finally, when you are done with you task, call *Application.stopProgress()* which hides the progress bar again.

The code snippet below illustrates the use of a progress bar. The bar is updated once a second for ten seconds. It makes use of the *startTimer()* function to start the delayed execution of the updating routine.

```
progress = 0;

function progressDemo() {
    progress = 0;
    app.startProgress ("Please wait", "Starting");
    startTimer ("doProgress()", 1000);
}

function doProgress() {
    progress += 0.1;
    if (!app.setProgress (progress, "Progress: " + progress*100 + "%"))
        app.stopProgress(), writeln ("Aborted!!");
    if (progress > 1)
        app.stopProgress();
    else
        startTimer ("doProgress()", 1000);
}
```

Localization

The Adobe GoLive SDK provides certain facilities which makes it easy to localize an extension. All messages, labels, window titles and menus normally display their text as defined in their tags. JavaScript messages are displayed as the script puts them together.


All of these texts and messages may be predefined to use different languages if desired. The SDK is capable of detecting the language in which Adobe GoLive runs. A module may contain a table which again contains translations for the strings in this module. This table is the content of the *jsxlocale* tag, which has no other function than to define a translation table.

The table itself needs to be set up in a well-defined way. The top row contains the language codes for each table columns. These codes are equivalent to the ISO-3166-1 language codes used widely in the Internet. For English messages, the country code "US" is used (all English speaking countries, please forgive this simplification). For German messages, "DE" is used, "FR" stands for French and so on. Since the default for all messages inside GoLive is English, the leftmost column must contain the English messages together with the country code "US".

A typical translation table would look like the one [Table 7.1](#) depicts.

TABLE 7.1 *Translation table example*

US	DE	NO
Are you sure?	Sind Sie sich sicher?	Er du sikker?
Search	Suchen	Start søk



When the extension module loads, it first loads this translation table. On each string the module loads, the contents of the string are checked against the US definitions in the table and the locale ID found inside GoLive. If a match is detected, the string is replaced. In above example, a button with the following definition

```
<jsxcontrol type="button" name="srch" value="Search">
```

would display "Suchen" on a German version of GoLive and " Start søk" on a Norwegian version. It should mentioned that the SDK removes all leading and trailing whitespace from the messages, and that the message comparison is case-sensitive. If you would have used "search" instead of "Search" in above table, the button would not have been translated.

When a script wants to used localized messages, it can use the method *Module.localize* (*message*) to translate a message. If a script wanted to confirm an action, it would, using the above table, use the following code:

```
if (confirm (module.localize ("Are you sure?")) ...) ...
```

This would return "Sind Sie sich sicher?" on a German GoLive. To get the current country code, a script can check the property *Module.locale*. This property is read/write, but assigning a different country code only changes the behavior of the *localize()* method within the affected module; menus or dialog contents are not changed.

For testing purposes, the GoLive setting for the country code can be overridden by specifying the *locale* attribute within the *jsxmodule* tag. The country code set there is valid for all menu, dialogs, and messages within the module.

When the GoLive SDK cannot find a translation for a given string, it attempts to find the string within the string database of GoLive itself. Therefore, many strings may be auto-translated without having to define them inside the module.

If desired, any translation can be turned off by setting the *locale* attribute to *NONE*.

For code examples of localization features, see the **Custom Box** and **Menus and Dialogs** samples .



Draft

8

Debugging

This chapter describes the JavaScript debugging tools that the SDK provides:

- The [Debugger Object \(\\$\)](#) is the source debugger's agent in the JavaScript runtime environment. This object provides information about the current JavaScript environment and manipulates breakpoints programmatically.
- The [JavaScript Shell Palette](#) provides a command line and output view you can use to execute JavaScript code interactively in the scope of a specified module.
- The [Integrated JavaScript Source Debugger](#) is a fully-featured debugger that provides a detailed breakpoints listing, step tracing, an interactive JavaScript command line, and the ability to edit live JavaScript objects from within breakpoints.

Integrated JavaScript Source Debugger

By default, all debugging services are disabled. If no module enables the integrated debugger, the \$ object, the JavaScript Shell palette and the integrated debugger are not available.

Enabling The Integrated Debugger and Other Debug Services

Each extension that is to use the integrated debugger must add the valueless `debug` attribute to its `<jsxmodule>` tag, as in the following example.

```
<jsxmodule name=myCoolExtension debug>
```

The `debug` attribute activates the integrated debugger for its own module only; however, once any module activates the integrated debugger, the JavaScript Shell palette and the \$ object become available to all extensions, even those that do not specify the `debug` attribute.

A module that activates the integrated debugger displays the [Script Debugger Window](#) shown in [Figure 8.1](#) automatically whenever GoLive encounters a runtime error or a breakpoint in any of the module's scripts. When GoLive encounters the error or breakpoint, script execution halts and the [Script Debugger Window](#) displays the current line of Javascript source and the current stack trace.

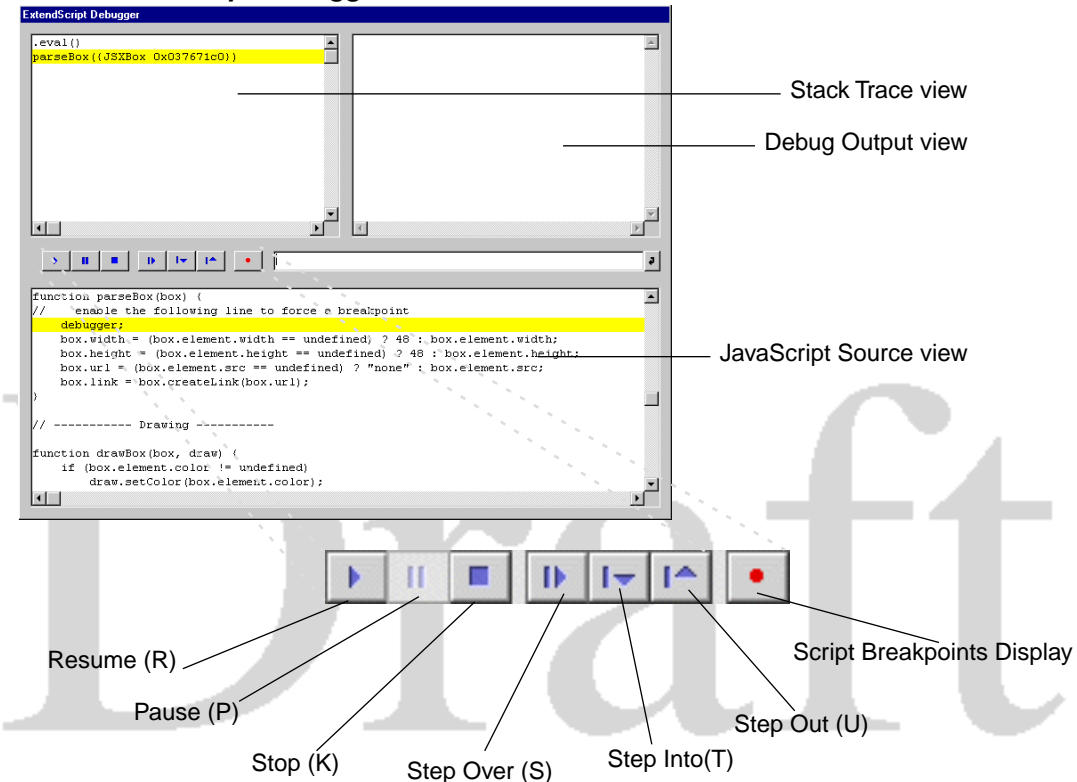
Script Debugger Window

This section describes the information and controls that the main Script Debugger window provides.

Viewing Debug Information

The GoLive Script Debugger window provides three informational views that [Figure 8.1](#) depicts.

FIGURE 8.1 Script Debugger window



The current stack trace appears in the upper-left pane of the script debugger window. This **stack trace view** displays the calling hierarchy at the time of the breakpoint. Double-clicking a line in this view changes the current scope, enabling you to inspect and modify scope-specific data.

All debugging output appears in the upper-right pane of the script debugger window. Specifically, output from the print method of the \$ object appears in this **debug output** view.

The currently-executing JavaScript source appears in the lower pane of the script debugger window. Double-clicking a line in this **JavaScript source view** sets or clears an unconditional breakpoint on that line; that is, if a breakpoint is in effect for that line, double-clicking it clears the breakpoint, and vice-versa.

Controlling Code Execution in the Script Debugger Window

This section describes the buttons that control the execution of code when the Script Debugger window is active. Most of these buttons also provide a keyboard shortcut available as a **Ctrl**-key combination on Windows platforms or a **Cmd**-key combination on Mac OS platforms.

**Resume**

Cmd-R (Mac OS)
Ctrl-R (Windows)

Resume execution of the script with the script debugger window open. When the script terminates, GoLive closes the script debugger window automatically. Closing the debugger window manually also causes script execution to resume. This button is enabled when script execution is paused or stopped.

**Pause**

Cmd-P (Mac OS)
Ctrl-P (Windows)

Halt the currently-executing script temporarily and reactivate the script debugger window. This button is enabled when a script is running.

**Stop**

Cmd-K (Mac OS)
Ctrl-K (Windows)

Stop execution of the script and generate a runtime error. This button is enabled when a script is running.

**Step Into**

Ctrl-T (Mac OS)
Cmd-T (Windows)

Halt after executing a single JavaScript statement in the script or after executing a single statement in any JavaScript function the script calls.

**Step Over**

Ctrl-S (Mac OS)
Cmd-S (Windows)

Halt after executing a single JavaScript statement in the script; if the statement calls a JavaScript function, execute the function in its entirety before stopping.

**Step Out**

Ctrl-U (Mac OS)
Cmd-U (Windows)

When the debugger is paused within the body of a JavaScript function, clicking this button resumes script execution until the function returns. When paused outside the body of a function, clicking this button resumes script execution until the script terminates.

**Script Breakpoints Display**

(no keyboard shortcut)

Clicking this button displays the [Script Breakpoints Window](#) shown in [Figure 8.2](#).

Using the JavaScript Command Line Entry Field

You can use the Script Debugger window's command line entry field to enter and execute Javascript code interactively within a specified stack scope. Commands entered in this field execute with a timeout of one second. If a command takes longer than one second to execute, GoLive terminates it and generates a timeout error.



Command line entry field. Enter in this field a JavaScript statement to execute within the stack scope of the line highlighted in the Stack Trace view. When you've finished entering the JavaScript expression, you can execute it by clicking the command line entry button or pressing the Enter key.

Command line entry button. Click this button or press Enter to execute the JavaScript code in the command line entry field. GoLive executes the contents of the command line entry field within the stack scope of the line highlighted in the Stack Trace view.

The command line entry field accepts any JavaScript code, making it very convenient to use for inspecting or changing the contents of variables. For example, in the display that [Figure 8.1](#) depicts, you could inspect and set properties of the `box` object, such as its width.

NOTE: To list the contents of an *object* as if it were JavaScript source code, enter the `object.toString()` command.

Setting Breakpoints

You can set breakpoints in the debugger itself, by calling methods of the `$` object, or by defining them in your JavaScript code.

Setting Breakpoints In the Script Debugger Window

When the GoLive Script Debugger window is active, you can double-click a line in the source view to set or clear a breakpoint at that line. Alternatively, you can click the BP button to display the Script Breakpoints window and set or clear breakpoints in this window as described in [“Setting Breakpoints in the Script Breakpoints Window”](#) and in [“Clearing Breakpoints in the Script Breakpoints Window”](#) on page 114.

Setting Breakpoints in JavaScript Code

Adding the debugger statement to a script sets an unconditional breakpoint. For example, the following code causes GoLive to halt and display the script debug window as soon as it enters the `parseBox` function.

```
function parseBox(box) {  
  // break unconditionally at the next line  
  debugger;  
  box.width = (box.element.width == undefined) ? 48 : box.element.width;  
  box.height=(box.element.height==undefined) ? 48 : box.element.height;  
  box.url = (box.element.src == undefined) ? "none" : box.element.src;  
  box.link = box.createLink(box.url);  
}
```


To set a breakpoint in runtime code, call the `$.setbp()` method, as the following example does.

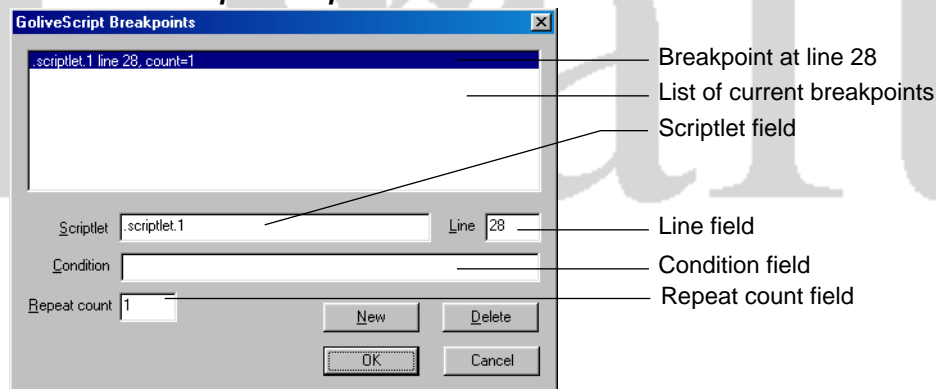
```
function parseBox(box) {  
    box.width = (box.element.width == undefined) ? $.setbp() : box.element.width;  
    box.height=(box.element.height==undefined) ? $.setbp() : box.element.height;  
    box.url = (box.element.src == undefined) ? $.setbp() : box.element.src;  
    box.link = box.createLink(box.url);  
}
```

This example breaks if any of the width, height, or src attributes of the custom element are undefined. Of course, you wouldn't put setbp method calls into commercial code—it's more appropriate for shipping code to set default values for undefined properties, as the previous example does.

Script Breakpoints Window

This section describes the information and controls that the Script Breakpoints window provides. Display of the Script Breakpoints window is controlled by the **Script Breakpoints** button in the main [Script Debugger Window](#) described on [page 109](#).

FIGURE 8.2 *Script Breakpoints window*



This dialog displays all defined breakpoints.

This dialog does not display

- Breakpoints defined by the debugger statement in JavaScript code.
- Temporary breakpoints.

The Script Breakpoints window provides the following controls:

- The **Scriptlet** field shows the name of the scriptlet that defines the breakpoint. This is the name used when the scriptlet was added to the engine.
- The **Line** field contains the line number of the breakpoint within the scriptlet.
- The **Condition** field may contain a Javascript expression to evaluate when the breakpoint is reached. If the expression evaluates to `false`, the breakpoint is not executed.
- The **Repeat count** field contains the number of times that the breakpoint must be reached before it is actually executed.

Breakpoints set in this window persist across multiple executions of a script.

Setting Breakpoints in the Script Breakpoints Window

Take the following steps to set a breakpoint in the [Script Breakpoints Window](#):

1. Enter a scriptlet name in the Scriptlet field
2. Enter a line number in the Line Number field.
3. Optionally, enter a condition such as `(i>5)` in the Condition field.
4. Optionally, enter in the Repeat Count field the number of times the specified line must execute before GoLive breaks into the debugger.
5. Click the OK button to set the breakpoint

Editing Breakpoints in the Script Breakpoints Window

Clicking a breakpoint in the breakpoints list copies its data into the edit fields at the bottom of the **Script Breakpoints** window. You can modify the contents of any of these fields to edit values in running code:

Clearing Breakpoints in the Script Breakpoints Window

Take the following steps to clear a breakpoint in the [Script Breakpoints Window](#):

1. Select the breakpoint from the breakpoints list.
2. Press the Delete key.
3. Click OK to accept the changes and dismiss the Script Breakpoints window..

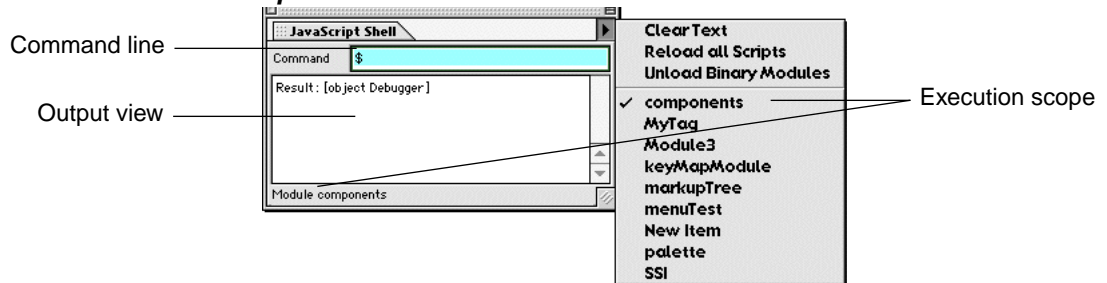
Debugger Object (\$)

The [\\$ Object \(Debugger Object\)](#) is present at all times. It provides properties and methods you can use to debug your JavaScript code; for example, you can call its methods to set or clear breakpoints programmatically, or to change the language flavor of the script currently executing in GoLive. It also provides properties that hold information about the version of the host platform's operating system and the flavor of JavaScript currently in use by GoLive Extend Script.

JavaScript Shell Palette

When at least one module enables debugging services by providing the valueless `debug` attribute in its `<jsxmodule>` tag, the **Window** menu contains the **JavaScript Shell** palette that [Figure 8.3](#) shows.

FIGURE 8.3 JavaScript Shell Palette



The **JavaScript Shell** palette provides a command line you can use to execute JavaScript code in a specified scope interactively.

The **JavaScript Shell** palette also provides an output view JavaScript code can use to display data. Your extension can call the `write` or `writeln` global functions to display data in this view. The `clearOutput` global function erases the contents of the output view.

The **JavaScript Shell** palette also provides a palette menu that always holds at least three entries:

Clear Text erases the contents of the output window.

Reload Scripts reloads the Javascript code of all currently-loaded extension modules. This command never reloads static data, which is defined in the `Main.html` file by tags like `<jsxmenu>`. It reloads only the contents of `<script>` tags. Objects and variables currently defined within the Javascript engine are unaffected by this command. This command does not call the `initializeModule` global function. To reload an extension completely, you must quit GoLive and restart it.

Unload Binary Modules unloads all binary modules. Subsequently, GoLive reloads each binary module on demand when a function in the module is called from JavaScript.

The palette menu is also populated with menu items representing currently-loaded extension modules. Selecting a module name from the palette menu specifies the module scope (extension) in which to execute the contents of the JavaScript command line.

NOTE: If the JavaScript shell window does not respond to command line input, look for the name of the current module in the lower-left corner. If it does not appear there, no modules currently enable debugging services. Debugging services are enabled only when at least one Extend Script module contains the `debugger` statement or the `debug` attribute to `<jsxmodule>` tag.



Draft

Part II

Reference

Draft



Draft

This chapter provides reference information for all of the markup tags the GoLive Extend Script SDK provides.

Modules

jsxmodule

`<jsxmodule>`

This optional tag defines the name of the Javascript module. This name can be checked as the Javascript variable `module.name`. The `name` attribute is responsible for defining that name. If the tag or the `name` attribute is missing, the module name is the name of the folder where the `main.html` file of the module is located. The `debug` attribute, when present, enables the integrated debugger. The JavaScript statement debugger; acts as a breakpoint when the integrated debugger is active.

The tag has the following attributes:

<code>name</code>	the Javascript name of the module.
<code>debug</code>	if this attribute is present, the module is in debug mode. The Javascript Output palette window is enabled, as well as the Javascript statement debugger
<code>timeout</code>	Number of seconds the JavaScript engine waits for a response from the currently-running JavaScript code. If this amount of time elapses without a response from the currently-running JavaScript code, GoLive generates a timeout runtime error, stops waiting for a response, and exits the script. By default, the GoLive engine waits indefinitely for JavaScript code to return. Your module can specify the amount of time GoLive waits for its JavaScript code to return by setting this attribute to a number of seconds between 0 and 9999. Setting this attribute to 0 seconds or a boolean value of <code>false</code> causes GoLive to wait indefinitely for this module's code to return. GoLive always waits indefinitely for its calls to your Event-Handling Functions to return. The code that sets the global default timeout value runs once when GoLive initializes all the modules in the Extend Scripts folder at startup time
<code>locale</code>	Defines a locale which should be used to translate all strings inside the module. This value overrides the locale setting of GoLive itself. Valid values are ISO-3166-1 country codes as used as Internet top level domains, like DE for Germany or IT for Italy. For all flavors of English, US is used. Although mainly usable for testing purposes, all auto-translation features inside the module can be disabled by setting the value to NONE.

Locales

jsxlocale

```
<jsxlocale> localizationTable </jsxlocale>
```

This tag is a container for a table which defines translations for the strings an extension uses in menu items, dialogs, and so on. For more information, see [“Localization” on page 106](#).

Dialogs

A dialog is an HTML form embedded within a `<jsxdialog>` tag. This form may hold a number of controls, each defined as `<jsxcontrol>` tags. When a dialog executes and the user changes the state of a control, GoLive calls the method `controlSignal()` with the affected control object as parameter. Here, the state of the control may be monitored and/or updated. A dialog is executed by calling its `runModal()` method, which invokes the dialog as a modal dialog. The return value of this method depends upon which button was clicked to close the dialog, or the argument supplied to the `exitModal` method.

The size of the dialog is either set in the width and height attributes of the `<jsxdialog>` tag or are extracted from the first embedded `<table>` tag. The positions of the controls are set by the `posx` and `posy` attributes of the corresponding `<jsxcontrol>` tag.

The user clicks a button to terminate a dialog. These buttons can have predefined names that GoLive recognizes as buttons which close the dialog, or they can dismiss the dialog by calling the `exitModal` method from their case in your implementation of the [controlSignal](#) method. Using "dialogok" defines the button to be the OK button. The name "dialogcancel" defines a button to be the Cancel button, and the name "dialogother" defines the button to be able to close the dialog without showing the specific behaviors of an OK or Cancel button. No other control can close a dialog.

Clicking a button with one of these specific names cause the `runModal()` method to return one of the following values:

dialogcancel 0

dialogok 1

dialogother 2

There are three possible different kinds of dialogs. The modal dialog is defined with the `<jsxdialog>` tag. A non-modal dialog is defined with the `<jsxpalette>` tag. This dialog can never be terminated. Finally, the `<jsxinspector>` tag creates an inspector dialog which is used together with custom drag-and-drop boxes.

Every control within a dialog is either accessible via its name as a property of the Dialog object or as part of the array controls of the Dialog object. All controls defined in the entire module are accessible via the global array controls as well.

jsxdialog

```
<jsxdialog name="objectName" title="Title Of Dialog" width="anInteger"
           height="anInteger" > </jsxdialog>
```

The `<jsxdialog>` tag wraps the dialog form. The dialog itself is a table, where the width and height attributes determine the size of the dialog window. Several controls may be placed within this grid by inserting `<jsxcontrol>` tags into the dialog. The dialog object is stored into the `dialogs` array of the document and may be accessed there. It contains the method `runModal()`, which causes the dialog to execute as a modal dialog. The name attribute defines the JavaScript name of the dialog.

To close the window and to terminate the dialog, a button with a special name needs to be created. When this button is clicked, the dialog is closed and a numeric value corresponding to the button name is returned by the `runModal()` dialog.

A dialog is either accessible via its name as a property of the global namespace or as a part of the global `dialogs` array.

The tag has the following attributes:

name the Javascript name of the dialog. The dialog is also part of the global `dialogs` array where it may be accessed via this name.

title the title of the dialog window (Windows only).

width the width of the dialog; may be overridden by an embedded `<table>` tag.

height the height of the dialog; may be overridden by an embedded `<table>` tag.

jsxpalette

```
<jsxpalette name="objectName" title="Title Of Palette" menu="anInteger"
            width="anInteger" height="anInteger" >
// to define a palette menu add <jsxmenu> and <jsxitem> elements here
</jsxpalette>
```

The `<jsxpalette>` tag creates a so-called palette window. This is a floating window like the Inspector window. Its title is displayed in the Window menu so the user can open and close the window like any other palette window. A palette window has an associated menu which is displayed as a pull-down menu in the upper right corner of the palette window.

A palette is either accessible via its name as a property of the global namespace or as a part of the global `dialogs` array.

The following attributes are defined:

name The Javascript name of the dialog. The dialog is also part of the global `dialogs` array where it may be accessed via this name. Since palettes are a global resource, use distinctive names if possible.

title The title of the palette window. This text also appears in the Window menu.

order The sort order of the menu. This is a number which indicates where in the Window menu this item should appear. The higher the number is, the lower down in the Window menu

the item appears. See the table in Appendix A for details. Values greater than 9999 cause undefined results.

jsxcontrol

```
<jsxcontrol
  type="KindOfControl" name="JavaScriptName"
  value="InitialValue" group="groupID"
  posx="NumOfPixels" posy="NumOfPixels" width="NumOfPixels"
  height="NumOfPixels" halign="SeeReference" valign="SeeReference" >
</jsxcontrol>
```

A control tag contains several attributes which define the position, alignment, and type of the control:

name the Javascript name of the control.

type the type of control:

button	pushbutton
checkbox	check box
check	check box
radiobutton	radio button
radio	radio button
edit	edit field which signals a change when the input focus changes if the edit field's contents have changed; that is, changing focus without changing content does not signal a change.
buttonedit	edit field that signals a change when the input focus changes and the contents of the field have changed. Optionally, this control can display an Enter button.
editarea	multiline edit field
static	static text field
color	color select field
urlgetter	URL entry field
popup	popup menu
list	list box
listbox	list box
custom	owner-draw control (see below)

valueThe initial value for all controls. Popup and list box controls accept a comma-separated list of items which they are able to display.

groupthe group ID for radio buttons. All buttons with the same ID are automatically turned off when one button is selected. If the attribute is omitted, the button is not affected by other buttons. The group ID is an arbitrary string value. It is valid for all controls within the same

extension; it would affect controls within other dialogs and the same group ID, so choose these IDs carefully.

`posx`, `posy` the upper left corner of the control in the dialog window. When the user drops the control on a table grid in layout mode, GoLive updates these attributes to reflect the position of the control on the page. This makes it easy to create f.ex. an inspector by dropping the controls on a table grid and moving them into position.

`width`, `height` the size of the control

`halign` the horizontal alignment of the control when the size of the window changes:

<code>left</code>	left-aligned
<code>center</code>	centered
<code>right</code>	right-aligned
<code>scale</code>	autoscaled

`valign` the vertical alignment of the control when the size of the window changes:

<code>top</code>	top-aligned
<code>center</code>	centered
<code>bottom</code>	bottom-aligned
<code>scale</code>	autoscaled

The alignment defines how the starting coordinates and the size of a control are affected when the size of the dialog changes. Thus, the starting coordinates of a control are always related to the predefined size of the dialog and should be computed relative to this size.

Custom controls may be coded entirely in Javascript (although this may be a bit tedious). To draw such a control, GoLive calls the method `drawControl()` with the control to draw whenever necessary.

When the user clicks a custom control, GoLive calls the Javascript method `mouseControl()`, again with the control that was clicked as the first parameter. There are three additional parameters:

`x`, `y` the location of the mouse cursor relative to the top left of the control

`mode` the mode. 0 means that the button was pressed, 1 means that the mouse was moved while the button was pressed, and 2 means that the mouse button was released. The code should call the control object's `refresh()` method to reflect a changed state due to the action of the mouse. When the mouse button was released, the Javascript method `controlSignal()` may be called as well as any other action.

When `mouseControl()` is not present, `controlSignal()` gets called instead when the mouse button is released over a custom control.

You can access any control in three ways

- by means of the `name` property of the Dialog object that contains it,
- as an element of the `controls` array of that Dialog object,
- by means of the global `dialogs` array.

Palette Items and Foreign Tags

The GoLive palette can easily be extended with new items which may be dragged and dropped like any other palette item. There are several tags which are needed to create a new palette. The `<jsxpalettegroup>` tag creates a new tab inside the palette which is then filled in with a number of `<jsxpaletteentry>` tags. Each of these tags is a placeholder for a custom box which is defined with the `<jsxelement>` tag. Finally, the `<jsxinspector>` tag is used to create a dialog which serves as the inspector for that specific box. The necessary icons for the palette as well as necessary pictures to draw the box may be defined using the widely known `` tag.

When a palette with the corresponding entries has been defined and the user drags one of these entries into the document, GoLive creates an empty box and places this box into the document. In order to draw the box, GoLive calls the Javascript method `drawBox()` with the `Box` object to be drawn as parameter. This method may use the global `Draw` object to draw its contents.

When the box has been resized, either by dragging at the box's borders or by entering data into the inspector dialog, GoLive calls the Javascript method `boxResized()`. This method receives three parameters, the box object itself, the new width, and the new height. The method should update the markup elements of the box besides of doing eventual calculations by itself.

Finally, the extension needs to set up the box when a HTML document containing the box has been read. To do so, GoLive calls the `parseBox` function, passing the box object as this function's argument. This method should check the markup elements of the box and adjust the look and feel of the box according to the settings of its markup values.

It is also possible to support non-HTML tags like ASP or PHP3 tags. The *type* attribute of the `<jsxelement>` tag allows you to specify the character that acts as tag delimiter. For example, if you specify `%` as the value of your `jsxelement` tag's *type* attribute, the SDK considers any string beginning with the `%` character to be a tag. Additional *type* values are:

- container* or *binary* for boxes containing more HTML,
- bracket* for bracketed tags like `[hello]`,
- percent* for tags enclosed in double percent signs like `%%hello%%`,
- ssi* for Server Side Includes,
- plain* for a standard tag.

The first word of the tag content is recognized as tag name. For example, the following declaration defines a handler for the SSI `#include` tag.

```
<jsxelement tagname="include" classid="someID" type="ssi">
```

This would cause GoLive to create a visual "include" box for a tags with the following content:

```
<!--#include virtual= http://www.adobe.com/test.js -->
```

When using nonstandard tag delimiters, however, access to the tag attributes as properties of the markup element is not possible, since these tags may contain any content. The content of these tags have to be interpreted by hand by calling the `getInnerHTML()` method of the markup element to retrieve the associated ASCII text.

The utility method `split()` of a markup element provides a similar functionality as the standard method `String.split()`. It splits the inner HTML into an array of strings, using either the blank or a supplied character as the word separator. Unlike `String.split()`, however, words in single or double quotes are recognized as strings. Use the method `Array.join()` to concatenate the contents of this array into a string which then can be used to set the inner HTML of the tag

To change the settings of the Web database, which controls how GoLive treats foreign tags, the Application object provides the properties `scanBrackets`, which allows to turn the scanning of [tags] on or off, `symmetricTokens` and `asymmetricTokens` which define which characters the GoLive scanner recognizes together with the `<` character as tag separators. Changing these settings change the settings inside the Web database; the settings are not saved unless someone explicitly opens the Web database and saves its contents.

jsxpalettegroup

```
<jsxpalettegroup
  name="objectName" display="tabName"
  tabOrder="anInteger" order="anInteger" >
  picture="tabIcon"
</jsxpalettegroup>
```

The `<jsxpalettegroup>` tag defines a new tab in the Objects palette. It has several attributes:

name the name of the palette group. This is either one of GoLive's predefined names or the name of a new palette group (case sensitive). The predefined names are:

Basic basic elements (images, plug-ins, scripts, and so on.)

Head head items

Frames frame elements

Custom custom items

Forms form elements

Project Site elements

CSObjects actions

WO WebObjects

display the name displayed to the user

taborder a number that describes the position of the tab for the palette group. The higher this number is, the further to the right the tab will be placed. This number may be omitted if a predefined group name was selected. For a listing of the `taborder` values of built-in tabs, see [“Objects Palette Entries” on page 196](#).

order the display order of the element within the selected palette. The higher the number, the further right the element will be displayed. The highest valid value is 32767.

picture The name of an image to be used as icon in the tab. This icon is scaled to a size of 12x12 pixels.

jspxpalettentry

```
<jspxpalettentry
    display="DescriptionOfTag" classid="customTagName"
    picture="paletteIcon">
    <customTagName
        HTMLContentOfCustomTagHere>
</jspxpalettentry> // end of custom palette entry
```

This tag defines a palette entry within the palette group. The attributes of the <jspxpalettentry> tag are:

display	the name of the element displayed to the user.
classid	the class name of the markup element.
picture	the icon which is displayed in the palette. This icon is scaled to a size of 24x24 pixels.

In your implementation of this tag, the *<customTagName HTMLContentOfCustomTagHere>* text is replaced by the exact HTML content this palette entry is to insert into the GoLive document. This content is a custom tag defined by the <jsxelement> tag. The jsxelement tag defines only the name of your custom tag; the default attributes your custom tag provides, as well as their values, are defined by this jsxpalettentry element. If the tag contains width and height attributes, Go Live uses them to set the size of the box it creates to represent the custom element.

img

```
<img name= JavaScriptName src=urlOrPathname>
```

The Extend Script SDK provides its own version of the widely-known tag. The SDK adds two attributes which are used to provide O/S specific loads of images.

name the name of the image

src the URL of the image source. If the winsrc (for Windows) or the macsrc (for the Macintosh) attribute is not present, this URL is used to load the image.

winsrc the URL of the image source in a Windows environment.

macsrc the URL of the image source in a Macintosh environment.

jsxelement

```
<jsxelement
    tagName="nameOfCustomTag" classid="jspxpalettentryID" type="seeBelow"
    leftMargin="numOfPixels" rightMargin="numOfPixels"
    topMargin="numOfPixels" bottomMargin="numOfPixels"
    invisible="boolean" fixedWidth=numOfPixels fixedHeight=numOfPixels >
</jsxelement>
```

This tag is used to describe the markup element which the palette entry can drag. It is located inside the document body along with any `` or `<script>` tags. This tag has a few attributes which control the type of box being displayed for the element. These attributes are not to be confused with the attributes of the custom tag the `<jxsxelement>` tag defines—the attributes of the custom tag are defined by the `<jspxpalettentry>` element that specifies the content the custom tag adds to the page.

tagname	the name of the tag that this <code><jxsxelement></code> defines.
classid	unique identifier associated with the tagname tag. This value must match the classid attribute of the <code><jspxpalettentry></code> and <code><jsxinspector></code> tags associated with the custom tag this <code><jxsxelement></code> tag defines.
type	The tag type, which is one of the following: <ul style="list-style-type: none"> binary a binary tag which may contain any HTML code. The code is not displayed and must be maintained programmatically. container A binary tag which may contain any HTML code. The code is displayed inside the box, which should have some margins set. The <code>drawBox()</code> function draws the box margin only. plain a standard tag. The optional attribute value denotes the width of the box ssi server side includes; a comment that begins with a pound sign (#) bracket a bracketed tag, like [hello] percent a tag surrounded with double percent signs x a tag beginning with <code><x</code>, where <code>x</code> is an arbitrary character.
leftMargin	the left margin of the generated container box.
rightMargin	the right margin of the generated container box.
topMargin	the top margin of the generated container box.
bottomMargin	the bottom margin of the generated container box.
invisible	the tag is invisible like f.ex. a comment tag. The menu command "Edit/hide invisible items" toggles the display of this tag on or off.
fixedwidth	if present, the box cannot be resized horizontally. This is the default for container boxes. This attribute can be used with or without a value; if a value is supplied, it denotes the width of the box.
fixedheight	if present, the box cannot be resized vertically. This is the default for container boxes. This attribute can be used with or without a value; if a value is supplied, it denotes the height of the box.

jsxinspector

```
<jsxinspector name="objectName" title="nameInWindowMenu"  
    classid="yourUniqueID" width="anInteger" height="anInteger" >  
    // jsxcontrol tags that provide inspector window controls go here  
</jsxinspector>
```

This tag defines the inspector dialog for the dropped box. Since an inspector simply is a special form of a dialog, its attributes, structure, and behavior is identical to the `<jsxdialog>` tag. Since all inspectors share the same window, the `<table>` tag which usually defines the window size can be ignored safely.

When a custom box is selected, GoLive activates its inspector window. Before displaying the window, GoLive calls the `inspectBox` method, passing the selected box as its argument. Your implementation of this method initializes the elements of the inspector with the current data.

When the user manipulates controls in the inspector, GoLive calls the `controlSignal` method just as it would for any other dialog, passing the control that changed as the argument to this method. Your implementation of this method alters the corresponding elements of the box as well as the HTML/XML representation of its code.

An inspector is either accessible via its name as a property of the global namespace or as a part of the global dialogs array.

The `<jsxinspector>` tag has the following attributes:

name the Javascript name of the inspector. The inspector is also part of the global dialogs array where it may be accessed via this name.

title the title of the inspector dialog.

classid the class name of the box which this inspector is responsible for. This name must match the `classid` attribute of the `<jsxpaletteentry>` and `<jsxelement>` tags.

Custom Element Example

The following example defines the tag `<showboat>`. This tag has the attributes `width`, `height`, and `image`, which defines an image to be displayed. The tag is represented by a plain box without any margins:

```
<jsxelement tagName="showboat" classid="boat" type="plain">
```

The box representing this tag should be draggable from a palette. We need, therefore, the following:

an icon 12x12 pixels for the palette tab

an icon 24x24 pixels representing the box to drag

The code to define the palette entry goes here:

```


<jspxpalettegroup name="ShowBoats" display="ShowBoats"
  taborder="3000" picture="tabIcon">
  <jspxpalettentry display="ShowBoat" classid="boat"
    picture="entryIcon">
    <showboat width="100" height="50" image="none">
  </jspxpalettentry>
</jspxpalettegroup>
```

You should define an inspector for the box. This inspector contains an URL getter control to alter the URL for the image attribute:

```
<jsxinspector name="insp" classid="boat" title="ShowBoat inspector"
width=100 height=100>
  <jsxcontrol name="image" type="urlgetter" posx=10 posy=10>
</jsxinspector>
```

Finally, we need to set up a number of Javascript methods. These methods should normally check for the right box class and control names, but this is omitted here for clarity.

parseBox()parses the HTML tag when the file is read.

drawBox()draws the box. Here, we draw the image or a circle if there is no image.

boxResized()reacts on changes of the box size. Here, we update the width and height attributes.

inspectBox()sets up the URL of the image in the inspector.

```
function parseBox (box, reason) {
  var value;
  value = box.element.width;
  box.width = value == undefined ? 100 : value;
  value = box.element.height;
  box.height = value == undefined ? 100 : value;
  box.url = box.element.image;
  box.link = box.createLink (box.url);
}
function drawBox (box, draw) {
  if (!box.pic)
    draw.frameOval (0, 0, box.width, box.height);
  else {
    box.pic.width = box.width;
    box.pic.height = box.height;
    box.pic.drawAt (0, 0);
  }
}
function boxResized (box, width, height) {
  box.element.width = width;
  box.element.height = height;
}
```

```

function inspectBox (box) {
    box.inspector.setLink (box.link);
    myBox = box;
}
function controlSignal (control) {
    var myBox = control.inspector.box;
    myBox.element.image = myBox.link.src;
    myBox.refresh();
}

```

Menus

There are three tags available to create menus. The `<jsxmenu>` tag defines a single menu. Its title is inserted into the menu bar right before the Window menu. Inside the `<jsxmenu>` tag, various `<jsxitem>` tags define the menu items inside the menu.


When a menu item defined with these tags is about to be displayed, GoLive passes the item to the `menuSetup` method. Your implementation of this method can enable or disable the item and set or clear its check mark. When the menu item is selected, GoLive passes the item to the `menuSignal` method as its first argument.

The following example defines two menus. The first menu is appended to the Special menu, while the second menu is inserted before the Window menu. The first menu item is initialized dynamically—its checked state changes each time the user chooses it. When a menu item has been selected, an alert window opens, displaying the menu item text.

```

<jsxmenubar>
  <jsxmenu name="special">
    <jsxitem name="one" title="Special One" testsignal>
    <jsxitem name="two" title="Special Two">
    <jsxitem name="three" title="Special Three">
  </jsxmenu>
  <jsxmenu name="more" title="More">
    <jsxitem name="more1" title="Alert One">
    <jsxitem name="separator" title="-">
    <jsxitem name="more2" title="Alert Two">
  </jsxmenu>
</jsxmenubar>
<script>
  function menuSetup (item) {
    // simply invert the check mark for "one"
    if (item.name == "one")
      item.checked = !item.checked;
  }
  function menuSignal (item) {
    if (item.name == "one")
      alert ("Selected: " + item.title);>
  }
</script>

```



A menu is either accessible via its name as a property of the global namespace or as a part of the global menus array. A menu item is again accessible either as a property of the Menu object it is defined within or as part of the items array of that Menu object.

jsxmenubar

```
<jsxmenubar></jsxmenubar>
```

The `<jsxmenubar>` tag opens the definitions of menus. It wraps the GoLive menu bar and should thus appear only once per document. Inside the `<jsxmenu>` tag, various `<jsxitem>` tags define the menu items inside the menu.

The `<jsxmenubar>` tag has no attributes.

jsxmenu

```
<jsxmenu name="name" title="Menu text"></jsxmenu>
```

The `<jsxmenu>` tag wraps a single menu inside the `<jsxmenubar>` tag. Each `<jsxmenu>` tag stands for exactly one menu, which GoLive inserts into the menu bar right before the Window menu. A `<jsxmenu>` tag contains one or more `<jsxitem>` tags, which define individual menu items.

The name attribute contains the Javascript name of the menu. There is, however, one special case which allows you to add menu items to the Special menu of the GoLive application by giving one of your menus the name "special". If GoLive detects a menu with the name attribute set to "special", it appends the contents of that menu to the Special menu rather than defining a new menu in the menu bar.

The title attribute contains the text that is presented to the user.

A menu is either accessible via its name as a property of the global namespace or as a part of the global menus array.

jsxitem

```
<jsxitem name="objectName" title="menuItemText" key = "hotKeyChars" dynamic>
```

The `<jsxitem>` tag defines a single menu item inside a menu. The name attribute contains the Javascript name of the menu, and the title attribute contains the text of the menu item. Setting the title attribute to a single dash "-" causes a separator item to be inserted.

The optional, valueless attribute `dynamic` indicates that the item needs to be checked and/or enabled according to the state of the application. When this attribute is present, GoLive calls the `menuSetup` method with the first parameter set to the menu item object. Inside that method, the item may be checked and/or enabled as needed. Menu items without the `dynamic` attribute are not called for initialization.

A menu item is accessible either as a property of the Menu object it is defined within or as part of the items array of that Menu object.

It is possible to define a hot key for a menu item. The value of the *key* attribute is a combination of the identifiers `Ctrl`, `Shift`, `Alt`, `Opt`, or `Cmd` as required.

- The `Cmd` identifier is valid on Mac OS platforms only; on Windows platforms, GoLive maps the *key* attribute value of `Cmd` to the `Ctrl` key. Similarly, the `Ctrl` identifier is valid on Windows platforms only; on Mac OS platforms, GoLive maps the *key* attribute value of `Ctrl` to the Command key.
- The `Opt` identifier is valid on Mac OS platforms only; on Windows platforms, GoLive maps the *key* attribute value of `Opt` to the `Alt` key. Similarly, the `Alt` identifier is valid on Windows platforms only; on Mac OS platforms, GoLive maps the *key* attribute value of `Alt` to the Option key.

These strings are connected with a plus sign and the hot key character. A valid string would be "`Shift+Ctrl+D`", for example. The string itself is displayed in the menu item as is on Windows; on the Mac, the corresponding hot keys are displayed.

Optionally, the ampersand `'&'` character may be used within the menu titles to force Windows to underline the character following the ampersand character. This character can then be used as hot key. On the Macintosh, these ampersand characters are removed from the string so they do not display. To display an ampersand, use two consecutive ampersands. When reading the title property, the ampersand characters are always removed from the original string to preserve compatibility across operating systems. Suppose you assign the string `"&New"` to the *title* property of a menu item. On Windows, it will display as `"New"`, while the Macintosh displays `"New"`. The value of the *title* property is `"New"`.

The menu item hot keys are visible in the Keyboard Shortcuts dialog and may be modified there. This is fine as long as the definition of menu items inside a module is entirely static. If, however, the order of the definition of menu items is changed, the Keyboard Shortcuts dialog loses the connection between the definitions it has stored on disk and the definitions found in the `<jsxitem>` tag. As a consequence, the dialog assigns the stored shortcut keys to the wrong menu items. Usually, this is not a big problem, since a finished module is used "as is" and not altered in any way. During development, however, this may lead to totally unwanted behavior. The only way to have the Keyboard Shortcuts dialog behave correctly after changing the order of menu item definitions is to erase the GoLive Preferences file. If the Keyboard Shortcuts dialog is not used, nothing happens, because the dialog is never activated.

This chapter describes the JavaScript objects that the SDK supplies. Some of these objects are created when GoLive interprets a `Main.html` file, while others, such as the application object, are always available.

Global Properties and Functions

These properties and methods are part of the global namespace. All dialogs, palettes, boxes, and menus are part of the global namespace and thus accessible via their name as long as the name follows Javascript guidelines. If more than one item have the same name, the results are unpredictable

Global Properties

app	<i>Application</i>	The application object. Read-only.
module	<i>Module</i>	The extension that this code belongs to. Read-only
document	<i>Document</i>	The current document. Read-only
documents	<i>Collection</i>	The array of documents. This array is read only. It may be indexed either by integers or by document name as reported by the <code>document.name</code> property. If the array contains more than one element of the same name, it cannot be predicted which element is returned.
boxes	<i>Collection</i>	The array of custom JSX boxes in the current document. This array is read only. It may either be indexed by integers or by name. If the array contains more than one element of the same name, it cannot be predicted which element is returned.
pictures	<i>Collection</i>	The array of pictures in the current document. This array is read only. It may either be indexed by integers or by name. If the array contains more than one element of the same name, it cannot be predicted which element is returned.
controls	<i>Collection</i>	The array of controls in the current document. This array is read only. It may either be indexed by integers or by name. If the array contains more than one element of the same name, it cannot be predicted which element is returned.
dialogs	<i>Collection</i>	The array of dialogs in the current document. This array is read only. It may either be indexed by integers or by name. If the array contains more than one element of the same name, it cannot be predicted which element is returned.

menus	<i>Collection</i>	The array of menus in the current document. This array is read only. It may either be indexed by integers or by name. If the array contains more than one element of the same name, it cannot be predicted which element is returned.
prefs	<i>Prefs</i>	The Preferences object gives extension modules access to their own preferences. All properties of this object are persistent and have the same value for every extension module.

Global Functions

These functions can be called by any object or function in the JavaScript namespace

fileGetDialog

```
fileGetDialog ([String prompt, String types])
```

This method presents the file open dialog that is standard for the platform on which GoLive is running. An optional string may be supplied with a prompt message. The type list differs on the Mac and the PC. On the Mac, it is a list of file types supplied as four character strings. The list is separated by commas. A valid type list would f.ex. be the list "TEXT,APPL", which would limit the display of files to text files and applications. On the PC, the list items are also separated by commas. Each item contains a list of file search masks, separated by the semicolon character. Optionally, an explanatory text may be appended with a colon. The string "*.jpg;*.jpeg:Images;*.html:HTML files" would create two entries in the selection list of possible file types, one for Images which displays all files ending with .jpg or .jpeg, and the other entry for all files ending with .html. The return value is a *File* object if a file was selected. If the dialog was cancelled, the return value is *null*.

Returns

[File Object](#)

filePutDialog

```
filePutDialog ([String prompt, String default, String typeList])
```

This method presents the file save dialog that is standard for the platform on which GoLive is running. The `prompt` message and the default file name to present to the user are required arguments. The third argument, which is a list of file types, is optional and is interpreted only on Windows platforms. This argument has the same structure as the type list which the *fileGetDialog()* function expects. The return value is a *File* object if a file was selected. If the dialog was cancelled, the return value is *null*.

Returns

[File Object](#)

alert

`alert (String text)`

Display an alert box with the given text and an OK button.

Returns

No return value.

confirm

`confirm (String text)`

Display a confirm box with the given text and Yes and No buttons. Returns *true* if the Yes button was clicked.

Returns

Boolean

prompt

`prompt (String prompt, String default)`

Display a dialog which allows the user to enter a line of text. The edit field is preset with the given default text. Returns the text the user entered or *undefined* if the user clicked the Cancel button.

Returns

String

write

`write (String text)`

Writes its arguments to the output view of the Javascript Output palette.

Returns

No return value.

writeln

`writeln (String text)`

Writes its arguments with a Linefeed appended to the output view of the Javascript Output palette.

Returns

No return value.

clearOutput

```
clearOutput()
```

Erases the contents of the output view of the Javascript Output palette

Returns

No return value.

absoluteURL

```
absoluteURL (String relURL, String baseURL, String separator)
```

Convert a relative URL to an absolute URL. Use the supplied base URL for the conversion. Optionally, a separator character may be supplied; if no separator is specified, the default separator character is the forward slash (/) character.

Returns

String

relativeURL

```
relativeURL (String absURL, String baseURL, String separator)
```

Convert an absolute URL to a relative URL. Use the supplied base URL for the conversion. Optionally, a separator character may be supplied; if no separator is specified, the default separator character is the forward slash (/) character.

Returns

String

createPicture

```
createPicture (String fileName)
```

Create a picture to be used during a drawing operation. Use the given file to load the picture from.

Returns

Picture

disposePicture

```
disposePicture (Picture pic)
```

Delete a picture created with createPicture(). The Picture object will be useless after calling this method.

Returns

No return value.

startTimer

```
startTimer (String scriptlet, Number timeout[, Boolean repeat])
```

Schedule a scriptlet for delayed execution. The given scriptlet will be run when the timeout elapsed. If the optional *repeat* parameter is *true*, the scriptlet is run repeatedly. The return value is a number which can be used as parameter for *stopTimer()* to stop a scheduled scriptlet.

Returns

Number

stopTimer

```
stopTimer (Number id)
```

Stop a scheduled scriptlet. The parameter is the return value of the *startTimer()* call used to schedule the scriptlet.

Returns

No return value.

startAction

```
startAction()
```

The *startAction()* function opens an undo action. All changes made to a custom box are recorded into this undo action.

Returns

No return value.

submitAction

```
submitAction()
```

The *submitAction()* function closes all recording and makes the undo action available to the undo/redo mechanism. This makes it possible to bundle several actions into one undo action.

Returns

No return value.

setActionName

```
setActionName()
```

The *setActionName()* function sets the text that the Undo/Redo menu item displays.

Returns

No return value.

GlobalPrefs Object

This object is available in the `app.prefs` property. It provides limited read-only access to GoLive's global preferences.

<code>imageFolder</code>	<i>String</i>	General: The Picture Import folder
<code>absoluteURLs</code>	<i>Boolean</i>	General: The Make new links absolute check box.
<code>writeGenerator</code>	<i>Boolean</i>	General: The Write "Generator Adobe GoLive" checkbox.
<code>scriptLibName</code>	<i>String</i>	Script Library: The name of the Script Library.
<code>scriptLibFolder</code>	<i>String</i>	Script Library: The name of the Script Library folder.

Prefs Object

Extension modules can create their own preferences which are maintained for all extension modules and are persistent across multiple runs of GoLive. This feature enables all extensions to share a common set of preferences. The preferences are accessed via the global object `prefs`. The properties of this object are all stored as preferences, so creating a new preference value is as easy as writing to a `prefs` property. If, for example, one module executed the code

```
prefs.present = "I am present";
```

All other modules can check for the presence of this module with the code

```
if (prefs.present == "I am present") ...
```

Application Object

The application object provides access to application global data and methods, like opening and closing documents.

Application Object Properties

The following properties of the Application object provide access to

- information about the application itself
- current user preferences
- file objects representing folders that the Application object uses

isModal	<i>Boolean</i>	Check whether the application currently is displaying a modal dialog. The property is true if this is the case. Read-only.
version	<i>String</i>	The version of GoLive. Read-only.
osVersion	<i>String</i>	The version of the operating system. Read-only.
scanBrackets	<i>Boolean</i>	This property controls whether the GoLive scanner recognizes bracketed tags like [hello]. It reflects the setting inside the Web database. The value of this property is not persistent unless the Web database is saved manually.
symmetricTokens	<i>String</i>	This property is a string containing all delimiters which the GoLive scanner uses together with the < character to recognize symmetric tags. By default, the string is set to ?%, which means that <%tags%> and <?tags?> are recognized. It reflects the setting inside the Web database. The value of this property is not persistent unless the Web database is saved manually.
asymmetricTokens	<i>String</i>	This property is a string containing all delimiters which the GoLive scanner uses together with the < character to recognize asymmetric tags. By default, the string is set to „!@“, which means that SGML <!tags> and <@tags> are recognized. It reflects the setting inside the Web database. The value of this property is not persistent unless the Web database is saved manually.
folder	<i>File</i>	A <i>File</i> object referring to the folder that holds the GoLive application. Read-only.
currentFolder	<i>File</i>	Returns a <i>File</i> object referring to the current folder. May be assigned a path name or a <i>File</i> object describing a folder.
settingsFolder	<i>File</i>	A <i>File</i> object referring to the folder that holds GoLive extension modules and other settings. Read-only.
systemFolder	<i>File</i>	A <i>File</i> object referring to the folder that holds the operating system. Usually, this is the C:\WINDOWS or C:\WINNT folder on Windows platforms; on Mac OS platforms, this folder can have any name and it can reside on any local disk.
tempFolder	<i>File</i>	A <i>File</i> object referring to the folder GoLive uses to store temporary files. Read-only.
trashFolder	<i>File</i>	A <i>File</i> object referring to the system Trash folder. On Windows platforms, this object refers to the local \RECYCLED directory. On Mac OS platforms, this object refers to the <i>startupDisk</i> :Trash folder. Read-only.
prefs	<i>GlobalPrefs</i>	This object provides limited read-only access to GoLive's preferences.

Application Object Functions

Functions of the application object provide a programmatic means of performing certain application-level tasks, such as passing keycodes and action codes to GoLive; creating new documents or opening existing ones; opening a URL in the current browser; accessing user preferences and configuration information; or quitting GoLive.

postKey(key)		Post a keycode to GoLive's input queue. The parameter is either the ASCII value of the key or a string whose first character is the character to post.
launchURL (String url)	<i>Boolean</i>	Launch the given URL in the current browser. If there is no current browser defined, this method does nothing and returns <i>false</i> .
quit()		Terminate the application by posting a quit message to the operating system. Same as the menu item File/Quit.
openMarkup (String docName)	<i>Document</i>	Open a document to access its markup tree without displaying a visual representation of the document on the screen. If this method opens the specified document successfully, its return value is the corresponding document object which has been appended to the <i>documents</i> array. On errors, the return value is <i>null</i> . The document remains invisible; only the markup tree is provided.
openDocument (String docName)	<i>Document</i>	Open a document. If no document name is given, the user is prompted for a file to open. If the document was opened, the return value is the corresponding document object which has been appended to the <i>documents</i> array. On errors, the return value is <i>null</i> .
newDocument()	<i>Document</i>	Open a new, empty document. If the document was opened, the return value is the corresponding document object which has been appended to the <i>documents</i> array. On errors, the return value is <i>null</i> . Same as the menu item File/New.
openPrefs()		Open the Preferences panel. Same as the menu item Edit/Preferences.
isModulePresent (String name)	<i>Boolean</i>	Check whether the module with the given name is enabled. Returns <i>true</i> if so, <i>false</i> otherwise.
startProgress (String title, [String message, Boolean doBusy, Number seconds])		Displays the progress or busy bar according to the setting of the <i>doBusy</i> parameter. The <i>seconds</i> parameter takes (optionally) a number of seconds before the window becomes visible. All parameters except for the first one are optional; by default, a progress bar is displayed.
setProgress (Number value, [String message])	<i>Boolean</i>	Update the progress bar. <i>value</i> is a value between 0 and 1. The optional <i>message</i> is displayed if supplied. On a busy bar, you can set the message; the progress value is ignored. The return value is <i>false</i> if the user clicked the stop button. This method should be called regularly during processing.
stopProgress()		Ends the display of the progress bar.

Document Object

The Document object corresponds to a GoLive document with its associated markup element tree.

Document Object Properties

The properties of the document object provide information about the current document, its elements, and the site that incorporates the page the document object represents.

type	<i>String</i>	The document type. This is currently one of the strings <i>markup</i> for documents containing an markup tree, <i>site</i> for site documents or <i>unknown</i> for all other documents. Read-only.
title	<i>String</i>	The document title. Read-only.
file	<i>File</i>	A <i>File</i> object representing the disk image of the document. If the document has not yet been saved before, this property is <i>null</i> . Use this object with great care, because GoLive may get confused if you f.ex. delete or rename an open document before saving it.
site	<i>SiteReference</i>	The root reference of a site document. This is usually a folder reference. The property is <i>null</i> for non-site documents. Read-only.
homePage	<i>SiteReference</i>	The home page reference of a site document. This is usually a file reference. The property is <i>null</i> for non-site documents. Read-only.
ref	<i>SiteReference</i>	Returns the Site Reference object associated with this document. This object is always present, regardless of whether the corresponding Site document is opened or not.
element	<i>Markup</i>	The markup tree of this document. If there is no tree (the document might be a simple text document, f.ex), the value is <i>null</i> . Read-only.
selection	<i>Selection</i>	The current user selection. This may either be a selected range or the current cursor position. Defined for markup documents only, <i>null</i> otherwise. Read-only.
encoding	<i>String</i>	The character encoding used for the document. It may be any encoding string usable in the META tag, like "iso-8859-1" or "shift_jis". Read-only.
history	<i>History</i>	The Undo/Redo history object allows you to inspect the currently defined number of Undo/Redo actions. Additionally, the current action may be selected, causing all necessary undo/ redo actions to be executed.

Document Object Functions

The functions of the document object enable you to create an undo object for the document it represents, as well as to save, close, reparse or reformat that document.

`save()` Save the document. If the document was not saved before, a dialog opens where the user can enter a file name.

`saveAs`
`(String fileName)` Save the document into a different file.

`close()` Close the document. If the document was not saved, the user is prompted whether he wants to save the document. If the document was saved successfully, the document object is invalid afterwards.

`reparse()` After manipulating the textual representation of a markup element, call this method to cause a complete reparsing of the document so the changes to the markup tree are reflected in the document layout view. All Javascript objects referring to the contents of the document, like boxes and markup elements, become invalid. For Site documents, this method reparses all pages in the site.

IMPORTANT: *Do not call this method from within the `parseBox` or `undoSignal` methods.*

`reformat()` This method works like the `reparse()` method. Additionally, it formats the document for better printed readability. For Site documents, this method operates on every page in the site.

IMPORTANT: *Do not call this method from within the `parseBox` or `undoSignal` methods.*

`createUndo (text)` *Undo* Creates an Undo object. The associated text is displayed in the History palette and in the Undo menu item.

Module Object

The Module object provides access to the extension module from JavaScript.

Module Object Properties

Properties of the module object provide the JavaScript name of the extension module and enable debugging services.

name	String	The name of the module. Read-only. This is either the value of the name attribute defined in the <code><jsxmodule></code> tag or the name of the folder where the main.html file is inside.
debug	Boolean	The value of the debug attribute of the <code><jsxmodule></code> tag. True if debugging is enabled, false otherwise.
locale	String	Returns the language in which GoLive runs. This value is a country code according to ISO-3166-1 as widely used in the Internet for top-level country domains, like DE for Germany or FR for France. For all flavors of English, US is used. The value is upper-case. Setting the value affects the way the <i>localize()</i> method works; its does not affect any menus or dialogs.

Module Object Functions

The module object defines only one function of its own, the `localize` function.

<code>localize</code> (String message)	String	This method attempts to translate the given string. It is assumed that the string is supplied in the English language. First, any table defined within the <i>jsxlocale</i> tag is scanned for the string. If found and a translated string is present which fits the current language setting, this string is returned. If the attempt fails, it tries to find the string within the GoLive translation database. If found, the translated string is returned. In this case, the setting of the <i>locale</i> property is ignored. If the string cannot be translated, the parameter itself is returned. Setting the <i>locale</i> property to "NONE" disables this method.
---	--------	--

Link Object

A link is an active URL which may be used to feed an *URLGetter* control in an inspector.

Link Object Properties

url	<i>String</i>	The URL that this link points to. It may be set as well as read. For dialogs and inspectors, this object is part of an <i>URLGetter</i> control.
local	<i>Boolean</i>	This property is <i>true</i> if the link points to a file residing on a local disk; it is <i>false</i> otherwise. Read-only.
mimeType	<i>String</i>	The MIME type of the link. Read-only.
protocol	<i>String</i>	The upload/download protocol of the link. Read-only.

Link Object Functions

drawIcon (Number x, Number y, [Number width, Number height])	Every link has an associated icon which reflects its current state. Use this method to draw this icon at a certain location. The icon may optionally be stretched to a given size.
linkChanged (link)	When the user changes a link, either in an <i>URLGetter</i> control or maybe in the Site View, GoLive calls this method with the affected link as parameter.

Box Object

GoLive creates a box object whenever the user drags a custom element's icon from the Objects palette to a GoLive document window, or when GoLive reads a document that contains a custom tag.

The box object is the visual representation of a custom markup element defined by the `<jsxelement>` and `<jsxpaletteitem>` tags. Each of these tags has a `classid` attribute that GoLive uses to identify the elements of the custom box. When these tags are used to define a custom box, each must specify the same `classid` attribute value. This value identifies the palette entry that creates the custom box and inserts HTML in the document, as well as the inspector window to display when the box is activated.

[“Box Object Functions” on page 146](#) describes the three callback functions that implement the functionality of the box.

Box Object Properties

name	String	The name of the box as defined by the <code>name</code> attribute of the <code><jsxelement></code> tag this box represents. Used to index the box in the <code>boxes</code> array. Read-only.
classid	String	The class name of the box as defined with the <code>classid</code> attribute. Read-only.
inspector	JSXDialog	A reference to the inspector dialog. This property is set only when the box is selected and an inspector dialog is active for that box and <code>null</code> otherwise. Read-only.
leftMargin	Number	The left margin of the box. Used for Container boxes only.
rightMargin	Number	The right margin of the box. Used for Container boxes only.
topMargin	Number	The top margin of the box. Used for Container boxes only.
bottomMargin	Number	The bottom margin of the box. Used for Container boxes only.
width	Number	The width of the box.
height	Number	The height of the box.
x	Number	The X position of the box relative to the top left corner of the document. Since this position is determined by the document layout, this property is read-only.
y	Number	The Y position of the box relative to the top left corner of the document. Since this position is determined by the document layout, this property is read-only.
element	Markup	The markup element associated with this box. Read-only.

Box Object Functions

<code>refresh()</code>		Request a repaint of the box.
<code>createLink</code> (String url)	<i>Link</i>	Create a new link and attach it to the box. If the URL is not supplied, an invalid link is created which may be set to an URL later.
<code>removeLink</code> (Link link)		Remove a link from the box. The link object becomes invalid after this call.

`parseBox (box, reason)`

GoLive calls this function when it reads the tag associated with this box object, either because the document containing the tag is being opened or because the user has switched back to a Layout view of this document from some other view. Your implementation of this function should read the contents of the tag and set up the box accordingly. When GoLive calls this function, it passes one of the following reason values:

- 1 the HTML document is being read.
- 2 the box has been dropped from the Objects palette or pasted from the clipboard.
- 3 the box is dragged around.

NOTE: Your implementation of the `parseBox` method must not call the `Markup.setInnerHTML()`, `Markup.setOuterHTML()`, `Document.reparse()`, or `Document.reformat()` methods. Attempting to call these methods returns a runtime error.

`drawBox`
(box, draw)

Global callback function called by GoLive to draw a custom box. The box to be drawn is supplied together with a *Draw* object which is used for the drawing operations.

IMPORTANT: *Your `drawBox` function must not call any functions other than drawing functions. An attempt to reparse the document or to download a file from within the `drawBox` method may cause GoLive to terminate abnormally.*

`boxResized`
(box,width,height)

Global callback function called by GoLive when a custom box has been resized. Should update the markup tag and/or recalculate the box.

`inspectBox (box)`

Global callback function called by GoLive when the inspector dialog for this box is about to be displayed. Should fill in the contents of the dialog to display the state of the box.

Collection Object

A collection is an array. Its members are indexed by name. In some collections, the members may be indexed by number as with a regular array.

Collection Object Properties

length	<i>Number</i>	The number of elements of the collection. This value is zero if the members cannot be indexed by number.
[index]	<i>Object</i>	The index operator retrieves the members of this collection. The members may be indexed by name; some collections may be indexed by numbers as well.

Collection Functions

The Collection object defines no functions of its own.

Picture Object

A picture is a graphics image, like a GIF or JPEG file, created by a tag. It is used for display purposes.

Picture Object Properties

name	<i>String</i>	The name of the picture as seen in the tag. Read-only.
width	<i>Number</i>	The width of the picture. Changing the size causes the picture to be stretched accordingly when drawn.
height	<i>Number</i>	The height of the picture. Changing the size causes the picture to be stretched accordingly when drawn.

Picture Object Functions

draw (Number x, Number y)	Draw the picture at the given location. This location is relative to the top left corner of its surrounding box or window.
------------------------------	--

Control Object

The Control object wraps a control created within a dialog with the `<jsxcontrol>` tag. Since this object may contain several different types of controls, a number of methods and properties exist which only deal with certain types of controls. These properties and methods return default values for controls they cannot handle. Setting values or calling methods is silently ignored for bad control types.

Control Object Properties

type	String	The type of the control as seen in the <code><jsxcontrol></code> tag in the document. Read-only.
name	String	The name of the control as seen in the <code><jsxcontrol></code> tag in the document. Read-only.
width	Number	The width of the control.
height	Number	The height of the control.
value	variable	The value of the control. This value depends on the type of the control. It may be a color value, the index of a selection, or the text of an edit field. For buttons and static text fields, this is the text which is displayed.
values	String	The contents of a list box or a popup menu as a string of comma-separated values. Setting this value with a comma-separated list like "One,Two,Three" reloads the entire contents of the control. For other controls, this property has the same effect as the <i>value</i> property.
enabled	Boolean	Get or set the enabled state of the control.
state	Boolean	Retrieve or set the selected state of the control. Setting this property causes the control to receive the input focus if possible.
selection	Number	Get or set the index of the currently selected item in a popup menu or list box. If the list box is multi-selectable, the property returns an Array object containing the index of all selected entries. When setting the property, a multi-selectable listbox turns on the selection for the given entry without deselecting the other entries. use the value -1 to deselect all entries for popups and list boxes. This property is only valid for popup menu controls and list boxes; all other controls return -1.
color	String	Get or set the color value for a color field control. Any valid HTML color may be used, like "red", or "#FF0000". This property is valid for color fields only; all other controls return an empty string as color value.

<code>itemCount</code>	<i>Number</i>	Retrieve the number of items for a popup menu. This property is valid for popup menus only. All other controls return 0. Read-only.
<code>multi</code>	<i>Boolean</i>	This property is valid for list boxes only. It contains a boolean indicating whether the user can select multiple entries or not. Setting this property to <i>true</i> allows for the selection of multiple list box entries. The <i>selection</i> property returns an <i>Array</i> object when multi-selection is turned on.
<code>group</code>	<i>Number</i>	This property is valid for radio buttons only. GoLive treats all radio buttons having the same <code>group</code> value as a group for the purposes of selection behavior; when one of the buttons in the group is selected, GoLive deselects the others in the group automatically.

Control Object Functions

<code>addItem</code> (String <i>text</i>)		Add an item to a menu. All other controls ignore this method. If the value of the <i>text</i> parameter is a single dash, this method adds a separator item to the menu.
<code>removeItem</code> (String <i>text</i>)		Remove an item from the menu control. This method is ignored for all other controls.
<code>removeAll()</code>		Remove all items from the menu control. This method is ignored for all other controls.
<code>setLink</code> (Link <i>link</i>)		Set the link for an URLGetter control. This method is ignored for all other controls.
<code>refresh()</code>		Initiate a repaint of the control. Useful for custom controls.
<code>beginDraw()</code>	<i>Draw</i>	Call this method to get a Draw object if you want to do direct drawing as a response to state changes. The <code>beginDraw()</code> call must be matched by a call to <code>endDraw()</code> to terminate the drawing. Calls to <code>beginDraw()</code> cannot be nested.
<code>endDraw()</code>		Matches the <code>beginDraw()</code> method. It ends any drawing operation and invalidates the Draw object returned by <code>beginDraw()</code> .
<code>controlSignal (ctl)</code>		Global callback function called by GoLive when the state of a control has changed due to user interaction.
<code>drawControl</code> (ctl, draw)		Global callback function called by GoLive when a custom control should be redrawn. The affected control is supplied along with a Draw object which is used for all drawing.

IMPORTANT: Your `drawBox` function must not call any functions other than drawing functions. An attempt to reparse the document or to download a file from within the `drawBox` method may cause GoLive to terminate abnormally.

mouseControl
(ctl,x,y,mode)

Global callback function called by GoLive when mouse is over the control and the button has been pressed (mode==0), the mouse has been moved with the button pressed (mode==1), or the mouse button has been released (mode==2).

Dialog Object

The Dialog class wraps a dialog definition made with the `<jsxdialog>`, `<jspxalette>`, or `<jsxinspector>` tag. It is part of the dialogs array in the global space, as is the dialog name.

Dialog Object Properties

name	<i>String</i>	The name of the dialog as seen in the <code><jsxdialog></code> tag in the document. Read-only.
title	<i>String</i>	The window title of the dialog. Applies only to modal dialogs.
focus	<i>Control</i>	Returns the Control object which currently has the input focus. Assigning a Control object to the property switches the input focus over to the given control.
box	<i>Box</i>	This property contains a reference to a box being inspected while an inspector dialog is active. It is <code>null</code> for all other dialogs and palettes, or when the inspector dialog is inactive. Read-only.
controls	<i>Collection</i>	The array of controls in this dialog. This array is read only. It may either be indexed by integers or by name. If the array contains more than one element of the same name, it cannot be predicted which element is returned.

Dialog Object Functions

runModal()	<i>Number</i>	Run this dialog as a modal dialog. The return value is set by pressing a button with a predefined name (See also the <code><jsxcontrol></code> tag) or by calling the <code>exitModal()</code> function. If you have specified a timeout for scripts, it is disabled while the dialog is visible. For more information, see “Setting the JavaScript Timeout” on page 33 .
exitModal (Number n)		Dismisses this dialog and causes <code>runModal()</code> to return the argument <code>n</code> .

Draw Object

The draw class is the wrapper for basic drawing. It works with an internal cursor which is used to draw lines and text. The coordinates are always relative to its surrounding box or window. This class is either supplied as a parameter to the drawBox()/drawControl() methods or returned by the beginDraw() method of the Control object.

Draw Object Properties

The Draw object provides no properties of its own.

Draw Object Functions

moveTo (Number x,
Number y)

Move the graphics cursor to a certain location.

lineTo (Number x,
Number y)

Draw a line from the current graphics location to the given position and move the graphics cursor to the new location.

setColor
(String color)

Set the drawing color. Any valid HTML color string may be used, like "red" or "#FF0000". Alternatively, three color for red, green, and blue may be supplied, where each color value ranges from 0 to 255.

frameRect
(Number x, Number y,
Number width,
Number height)

Draw an outlined rectangle.

fillRect
(Number x, Number y,
Number width,
Number height)

Draw a filled rectangle.

frameOval
(Number x, Number y,
Number width,
Number height)

Draw an outlined oval.

fillOval
(Number x, Number y,
Number width,
Number height)

Draw a filled oval.

invertRect
(Number x, Number y,
Number width,
Number height)

Inverts the colors of the given rectangle, producing an XOR effect. Calling this method twice undoes the effect of the first call.

penSize (Number width)	Set the width of the drawing pen.
textFont (String fontName)	Set the name of the font to use for text output. Use "ApplicationFont" to set the font to the application font.
textSize (Number size)	Set the size in points for the font to use for text output.
textFace (Number bits)	Set the attribute bits for the font to use for text output. These are bit values which may be added together: 1 – bold 2 – italics 4 – underlined 8 - outlined (Macintosh only) 16 - shadowed (Macintosh only) 32 - condensed (Macintosh only) 64 - extended (Macintosh only)
drawString (String text)	Draw a string at the current graphics cursor location.
stringWidth (String text)	Calculate the width of a string in pixels by using the current font settings.
stringHeight (String text)	Calculate the height of a string in pixels by using the current font settings.
getDrawInfo() <i>Number</i>	Retrieve a magic number which may be passed on to a native language extension if custom drawing is implemented there. May be casted to a JSADrawInfo structure.

Markup Object

The Markup class wraps a markup tag, a text block, or a comment. Its properties provide the tag name and the element type. The element contains the attribute array, and the array of subelements. This array may be searched for specific tags. The root element has no parent. The markup object has the ability to overwrite its textual representation.

The attributes of an element are implemented as properties. Setting a property causes the corresponding attribute to be written to the HTML code. Reading a property returns the property value or undefined if the attribute does not exist. Deleting a property deletes the attribute from the code.

Markup Object Properties

tagName	<i>String</i>	The tag name of this element.
tagStart	<i>String</i>	The starting sequence of this tag. This may f.ex. be "<", "[", "percent", or "<%".
symmetric	<i>Boolean</i>	This member is <i>true</i> if the end of the tag matches the beginning of the tag, like in "<%xxx%>".
elementType	<i>String</i>	Retrieve the type of the element. It may be one of the strings "tag", "text", "comment", or "bad".
parent	<i>Markup</i>	Retrieve the parent element of this element. The root element returns null. Read-only.
subElements	<i>Collection</i>	The array of sub elements. This array is read only. Its elements can be accessed by means of an integer index or the tagName property . If the array contains more than one element of the same name, it cannot be predicted which element is returned.

Markup Object Functions

getSubElementCount (String tagName)	<i>Number</i>	Count all sub elements for a specific tag and return the number of elements found. If the tag name is omitted, the total number of all subelements is returned.
getSubElement (String tagName, [Number index])	<i>Markup</i>	Get the nth sub element of a specific tag. The second argument is optional and defaults to 0. The first argument may be omitted to index all elements sequentially.
getOuterHTML()	<i>String</i>	Retrieve the HTML representation of the element, including the element tag itself.
getInnerHTML()	<i>String</i>	Retrieve the HTML representation of the element, excluding the element tag itself.
setOuterHTML (String text)		Replace the HTML representation of the element, including the element tag itself. Cannot be called inside parseBox().
setInnerHTML (String text)		Replace the HTML representation of the element, excluding the surrounding tag characters. On binary tags (tags with a matching end tag), the contents of the tag are replaced. Cannot be called inside parseBox().
split ([String sep])	<i>Array</i>	Split the contents of the inner HTML into an array of strings. Use the given character as a separator; if none given, use spaces as separators. Do not split quoted strings containing the separator character. Useful for checking the contents of special tags.

Menu Object

A menu is defined with a `<jsxmenu>` tag containing one or more `<jsxitem>` tags, one for each item. When a menu is about to be displayed and the dynamic attribute is present, the Javascript method `menuSetup (item)` is called for each menu item. The method may set the enabled and checked state for each item. When the user selects a menu item or types its keyboard shortcut, GoLive calls the `menuSignal` method, passing the selected menu item as its argument.

Menu Object Properties

name	<i>String</i>	The JavaScript name of the menu object, as specified by the name attribute of the <code><jsxmenu></code> tag that defines the menu. Read-only.
title	<i>String</i>	The title of the menu as displayed to the user. Use the ampersand '&' character to cause Windows to underline the character following the ampersand character and to accept that character as a hot key. On the Macintosh, ampersand characters are ignored and removed from the string. Use double ampersands to display an ampersand character. The property value is the title string with the ampersand characters removed.
selection	<i>Number</i>	Get or set the currently selected menu item.
items	<i>Collection</i>	The array of menu items. This array is read only. It may either be indexed by integers or by name. If the array contains more than one element of the same name, it cannot be predicted which element is returned when addressed by name.

Menu Object Functions

<code>addItem</code> (String name, String title, Number before)	Append a new menu item. An optional third parameter allows the item to be inserted before a specific other item.
<code>removeItem</code> (String name)	Remove a menu item.

MenuItem Object

The menu item corresponds to a `<jsxitem>` tag. It is always part of a `Menu` instance. When a menu is about to be displayed, the Javascript method `menuSetup (item)` is called for each menu item. The method may set the enabled and checked state for each item which has been flagged with the `dynamic` attribute. When the user selects a menu item or types its keyboard shortcut, GoLive calls the `menuSignal` method, passing as its argument the `jsxmenuItem` object that represents the selected menu item.

MenuItem Object Properties

name	<i>String</i>	The JavaScript name of the menu item object, as specified by the name attribute of the <code><jsxmenuItem></code> tag that defines the menu item. Read-only.
menu	<i>Menu</i>	The menu where this item belongs to. Read-only.
title	<i>String</i>	The menu item text this object displays in a menu. If the value of this property is a single dash, the menu item is a separator. Use the ampersand <code>'&'</code> character to cause Windows to underline the character following the ampersand character and to accept that character as a hot key. On the Macintosh, ampersand characters are ignored and removed from the string. Use double ampersands to display an ampersand character. The property value is the title string with the ampersand characters removed.
checked	<i>Boolean</i>	Access the check mark left of the menu item.
enabled	<i>Boolean</i>	Get or set the enabled state of the item.
dynamic	<i>Boolean</i>	Get or set the need to change the state of this item before it is displayed.

MenuItem Object Functions

<code>menuSetup (item)</code>	Global callback function called by GoLive when a menu item with the dynamic attribute is about to be displayed.
<code>menuSignal (item)</code>	Global callback function called by GoLive when a menu item has been selected.

Selection Object

The selection object is part of the document object. Most of its properties are read only, except for the element property. Assigning a markup element to this property causes the element to be selected if possible. Assigning Null or anything else than a markup element causes the current selection to be removed.

Selection Object Properties

type	<i>String</i>	The type of the selection: point No selection. The selection reflects the cursor position. part A part of the current markup element has been selected. This is true for simple text elements. full The whole markup element has been selected. This is f.ex. true when the user clicks an image box. complex The selection covers more than one markup element, or expands partially into other tags. outside The selection is outside of the current markup element. This is extremely unlikely to happen.
element	<i>Markup</i>	The first selected element. Assigning a Markup object to this property causes the corresponding element to be selected if possible. Assigning anything other than a Markup object causes the selection to be removed.
start	<i>Number</i>	The beginning offset of the selection, as relative to the outer HTML of the element. Read only.
length	<i>Number</i>	The length of the selection. Read only.
text	<i>String</i>	The selected text, according to the settings of start and length. Read only.

Selection Object Functions

The selection object defines no functions of its own.

Undo Object

The `createUndo` method of the [Document Object](#) creates an undo object associated with a specific user operation on a specific custom box. You must store in this object any properties your custom box needs to perform the operation, undo its effects, or redo its effects. Once all of these properties have been added to the undo object, you pass it back to GoLive by calling the undo object's `submit` method.

Whenever the user issues the command to perform the action, undo the action, or redo the action, GoLive passes the undo object to your extension's `undoSignal` method along with a code indicating whether your method should do, undo, or redo the operation the undo object encapsulates. Your implementation of this method utilizes the previously-stored properties to take appropriate action.

For a code example, see [“Supporting the Undo and Redo Commands” on page 95](#).

Global Undo Functions

In addition to the functions that the Undo object provides, you can use the following global functions to support undo/redone functionality:

- The `startAction()` function opens an undo action. All changes made to a custom box are recorded into this undo action.
- The `submitAction()` function closes all recording and makes the undo action available to the undo/redone mechanism. This makes it possible to bundle several actions into one undo action.
- The `setActionName()` function sets the text that the Undo/Redo menu item displays.

Undo Object Properties

The undo object has no properties of its own. You must add to it any properties your extension requires to perform an operation or reverse its effects. You can add properties to the Undo object simply by declaring and assigning them, as the following example does:

```
// create empty Undo object
var undo = document.createUndo ("operationName")
// add properties
undo.myProperty = myValue;
undo.myOtherProperty = myOtherValue;
undo.myThirdProp = yetAnotherValue;
// submit the undo object to GoLive
undo.submit();
```

Once you've finished adding properties to the undo object, return it to GoLive by calling the undo object's `submit` method.

yourPropertyName *String* Add as strings any properties you need to do, undo, or redo this action.

Undo Object Functions

submit()	The undo object is submitted to GoLive. GoLive in turn stores the object into its History list and calls the callback function described below.
undoSignal (undo, action)	<p>Callback function for Undo objects. GoLive calls this function for three purposes, which are reflected in the action parameter:</p> <ul style="list-style-type: none">0 – Do. Called as soon as submit() is called.1 – Undo.2 – Redo. <p>The methods Document.reparse() as well as Document.reformat() cannot be called within this function. An attempt to call these methods results in a runtime error.</p>

History Object

The History object is part of the *Document* object. It provides access to the Undo/Redo history.

History Object Properties

length	<i>Number</i>	The number of history entries. Readonly.
current	<i>Number</i>	The current index of the history. This is a number between 0 and the <i>length</i> property. Setting this value causes the history to set the current element to the given index, performing all necessary undo/redo actions.
maxCount	<i>Number</i>	This property reflects the maximum count of undo/redo actions. This is a value between 1 and 999. As opposed to setting the value in the History palette, setting the value here does not make the change permanent; it is only valid for the current program run.
[index]	<i>String</i>	Indexing the History objects returns the name of an Undo action. Readonly.

History Object Functions

The History object defines no functions of its own.

SiteReference Object

The SiteReference object is part of a site or markup document. Each reference stands for either a file, a folder, or other types of links contained in a file like email addresses. The root of the site can be found in the property document.site, while the home page is in document.homePage. The property document.ref returns a Site reference object for the current document, which may be used to traverse outgoing and incoming links even if the document is not part of a site document.

SiteReference Object Properties

All SiteReference properties are read-only.

name	<i>String</i>	The name, usually the file name or its URL.
type	<i>String</i>	The type of the reference. One of the following strings: html a HTML file folder a folder containing files alias a file or folder alias image an image file mail an email address invalid an invalid reference
siteDoc	<i>Document</i>	The Site document which this reference belongs to. This property is <i>null</i> if the Site document is not opened.
url	<i>String</i>	The URL of the reference as used in the document.
longUrl	<i>String</i>	The full URL of the reference.
anchors	<i>Array</i>	The array of anchors contained in this file. This array is empty if there aren't any anchors on the page.
local	<i>Boolean</i>	True if the reference is a file on local storage, false otherwise.
(name)	<i>String</i>	Add by name any properties you may need for your Undo. The properties are stored as strings.
status	<i>String</i>	The status of the reference, one of the following: error parsing error or other error empty an empty reference checking the reference is currently being checked invalid an invalid reference ok the reference is good and validated

lockStatus	<i>Number</i>	The locking status of the reference: 0 unknown 1 read only 2 read/write 3 checked in 4 checked out exclusively 5 checked out non-exclusively 6 broken
fileSize	<i>Number</i>	The physical size of a file, 0 for other types.
mimeType	<i>String</i>	The MIME type of the reference.
protocol	<i>String</i>	The upload/download protocol of the file.
title	<i>String</i>	The document title as defined in the <title> tag of the HTML document. Empty for other types.
prefs	<i>Prefs</i>	A preferences object. Here, you can get or set any preference data you might be interested in. The data is stored together with the reference and may be used for marking the reference or setting other attributes.

SiteReference Object Functions

getFiles (String type)	<i>SiteCollection</i>	Retrieve the files contained in a folder reference. The optional type argument may be a string containing a list of values as returned by the type property, separated by any non-alpha character, except for the mail value. For example, html+folder would be a valid string. The call works only when the Site document is open.
getIncoming (String type)	<i>SiteCollection</i>	Retrieve the list of SiteReference objects that point to this SiteReference object. The optional type argument may be a string containing a list of values as returned by the type property, separated by any non-alpha character, except for the folder value. For example, html+image would be a valid string. The call works even if no Site document is open; it works, however, only for opened documents, so it would return a reference only if another open document referenced this reference.
getOutgoing (String type)	<i>SiteCollection</i>	Retrieve the list of SiteReference objects this SiteReference object references. The optional type argument may be a string containing a list of values as returned by the type property, separated by any non-alpha character, except for the folder value. For example, html+image would be a valid string. The call works even if the Site document is not open.
show()		Displays and highlights the reference in the Site window.

<code>open()</code>	<i>Document</i>	Opens the reference in GoLive as a HTML document. Valid for HTML documents only. The returned Document object becomes the current document.
---------------------	-----------------	---

SiteCollection Object

The SiteCollection object is part of a site document. It contains a number of SiteReference objects. Despite their similar names, the SiteCollection object is not related to the Collection object, nor is it used similarly.

Site Collection Object Properties

The SiteCollection object defines no properties of its own.

Site Collection Object Functions

<code>first()</code>	<i>SiteReference</i>	Returns the first element of the collection. If the collection is empty, the result is null.
<code>last()</code>	<i>SiteReference</i>	Returns the last element of the collection. If the collection is empty, the result is null.
<code>next()</code>	<i>SiteReference</i>	Returns the next element of the collection. If there is no next element in the collection, the result is null.
<code>prev()</code>	<i>SiteReference</i>	Returns the previous element of the collection. If there is no previous element in the collection, the result is null.
<code>[index]</code>	<i>SiteReference</i>	If the name of a reference is known, it may be used to access that element directly within a collection.

File Object

The File object wraps operating system files and folders. To create a File object, call the `JSXFile` constructor function. When initialized with a partial file name, the File object always converts this partial file name to a full path name in order to create an unique reference to a certain file. File objects that refer to the same file are distinct objects; for example, the file object returned by the property `document.file` is different from the File object `document.homePage.file` even if both refer to the same disk file

File Object Properties

<code>name</code>	<i>String</i>	Returns the file name without the path. Readonly.
<code>path</code>	<i>String</i>	Returns the full path name of the file. Readonly.
<code>url</code>	<i>String</i>	Returns the full path name of the file encoded as URL. Readonly.
<code>parent</code>	<i>File</i>	The parent object of this file object. If the file is a top level object, the value is <i>null</i> .
<code>exists</code>	<i>Boolean</i>	Returns <i>true</i> if the file or folder exists on disk. Readonly.
<code>isFolder</code>	<i>Boolean</i>	Returns <i>true</i> if the file object is an existing folder. Readonly.
<code>macType</code>	<i>String</i>	Returns the file type of the file on a Macintosh as a four-character string. If the file type cannot be determined or on Windows, the return value is "????". Readonly.
<code>macCreator</code>	<i>String</i>	Returns the creator ID of the file on a Macintosh as a four-character string. If the file creator cannot be determined or on Windows, the return value is "????". Readonly.
<code>size</code>	<i>Number</i>	Returns the file size. For folder, the return value is 0. Readonly.
<code>eof</code>	<i>Boolean</i>	Returns <i>true</i> if an EOF condition has been raised during a read operation. Readonly.
<code>error</code>	<i>Boolean</i>	Returns <i>true</i> if an error has been detected during a read/write operation. Readonly.
<code>readOnly</code>	<i>Boolean</i>	Mirrors the Readonly/Locked flag of the file on disk. May be set to change the status of that flag.
<code>lastError</code>	<i>String</i>	Contains an explanatory text describing the last I/O error related to this file. This property is often used to check the result of a HTTP upload or download. May be set to any string value or to the empty string to clear the value.

File Object Functions

JSXFile (String path)	<i>File</i>	The constructor or global function for this object creates a new file object. The parameter may either be a pathname or a local URL. If the parameter is omitted, the object is initialized to the current working directory
createFolder()	<i>Boolean</i>	Attempt to create a folder at the location the file name points to. Returns <i>true</i> if the folder could be created.
getFiles ([String mask, String type])	<i>Array</i>	Get a list of files contained in the current folder object. The mask is the search mask for the file names. It may contain question marks and asterisks and is preset to <i>*</i> to find all files. The second, optional parameter is the Macintosh file type supplied as a four-byte string, like "TEXT". It is ignored on Windows systems. The return value is an array of File objects which correspond to the files found. The return value is <i>null</i> if the call is not used on a folder object.
rename (String newName)	<i>Boolean</i>	Rename the file object to the new name. The new name must not have a path. Returns <i>true</i> if the file object could be renamed.
move (String newPath)	<i>Boolean</i>	Move the file object to a different location. The parameter is the name of the new location; it can either be a partial pathname, a full pathname or a local URL. Returns <i>true</i> if the file was moved successfully. The File object is changed to point to the new location of the file.
copy (String newPath)	<i>Boolean</i>	Copy the file object to a different location. The parameter is the name of the new location; it can either be a partial pathname, a full pathname or a local URL. Returns <i>true</i> if the file was copied successfully. The File object remains unchanged.
openMarkup()	<i>Document</i>	Open a document to access its markup tree without displaying a visual representation of the document on the screen. If this method opens the specified document successfully, its return value is the corresponding document object which has been appended to the documents array. On errors, the return value is <i>null</i> . The document remains invisible; only the markup tree is provided.
openDocument()	<i>Document</i>	Open a document. If the document was opened, the return value is the corresponding document object which has been appended to the documents array. On errors, the return value is <i>null</i> .

open (String openMode)	<i>Boolean</i>	<p>Open the file for subsequent read/write operations. The supplied open mode is equivalent to the open mode of the C library call <i>fopen()</i>:</p> <p>rOpens for reading. If the file does not exist or cannot be found, the call fails.</p> <p>wOpens an empty file for writing. If the file exists, its contents are destroyed.</p> <p>aOpens for writing at the end of the file (appending); creates the file if it does not exist.</p> <p>r+Opens for both reading and writing. (The file must exist.)</p> <p>w+Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.</p> <p>a+Opens for reading and appending; creates the file if it does not exist. When a file is opened with the "a" or "a+" modes, all write operations are at the end of the file. The file pointer can be repositioned, but is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.</p> <p>tOpen in text (translated) mode. In this mode, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading/writing with "a+", OPEN checks for a CTRL+Z at the end of the file and removes it, if possible. Also, the system attempts to recognize CRLF sequences and converts them to Line Feed characters. This is the default mode.</p> <p>bOpen in binary (untranslated) mode; translations involving carriage-return and linefeed characters are suppressed. May be supplied along with one of the other opening modes.</p>
read (Number count)	<i>String</i>	Reads the contents of the file from the current position on. In translated mode, CRLF sequences are translated to Line Feeds. Returns a string which contains up to the number of characters which were supposed to be read.
readln()	<i>String</i>	Reads one line of text. In binary mode, line feeds are recognized as CR, LF or CRLF pairs.
write (String text, ...)	<i>Boolean</i>	Write the given string to the file. The parameters of this function are concatenated to a single string. Returns <i>true</i> if the string could be written successfully.
writeln (String text, ...)	<i>Boolean</i>	Write the given string to the file and append a Line Feed sequence. The parameters of this function are concatenated to a single string. Returns <i>true</i> if the string could be written successfully.
seek (Number pos)	<i>Boolean</i>	Seek to a certain position in the file. Returns <i>true</i> if the position was changed.
tell()	<i>Number</i>	Returns the current position in the file.

<code>close()</code>	<i>Boolean</i>	Close the file. If the file has been written to, its contents are flushed to disk. Returns <i>true</i> if the file could be closed successfully.
<code>get</code> (String remoteURL, [String mimeType])	<i>Boolean</i>	Download a file from a remote HTTP server. The URL is the name of the remote file to download. The file is stored into the location which the <code>JSXFile</code> object points to. If a file with the same name already exists, it will be overwritten. The result is <i>true</i> if the file was transferred successfully, otherwise it is <i>false</i> . The <code>lastError</code> property contains the HTTP status code after the transfer.
<code>put</code> (String remoteURL, [String mimeType])	<i>Boolean</i>	Upload a file to a remote HTTP server. The URL is the name of the remote location where the file is stored. The HTTP server must be able to fulfill HTTP PUT requests. The result is <i>true</i> if the file was transferred successfully, otherwise it is <i>false</i> . The property <code>lastError</code> contains the HTTP status code after the transfer.

\$ Object (Debugger Object)

The \$ object, also known as the debugger object, is always present. It provides properties and methods you can use to debug your JavaScript code.

\$ Object Properties

The properties of the \$ object provide information about the version of the host platform's operating system and the flavor of JavaScript currently in use by GoLive Extend Script.

<code>os</code>	<i>String</i>	Return the operating system version as a string.
<code>flavor</code>	<i>String</i>	Sets or returns the language flavor in use by GoLive Extend Script. Possible values are <code>Javascript</code> for the Netscape flavor, <code>Jscript</code> for the Microsoft flavor or <code>ECMAScript</code> for the ECMA-262 flavor. <code>Javascript</code> is the default value of this property.

NOTE: The `flavor` property applies only to GoLive. It does not apply to Web browsers.

\$ Object Functions

The functions of the \$ object

- set and clear breakpoints
- print messages to the main debugger window
- request garbage collection

<code>print(...)</code>	Prints its parameters to the Output window of the debugger UI. Does not activate the debugger when called.
<code>setbp</code> <code>(String</code> <code>scriptlet,</code> <code>Number line,</code> <code>[String eval,</code> <code>Number passes])</code>	Set a breakpoint. The scriptlet parameter is the name of a scriptlet as defined when the scriptlet was entered into the engine. The line number is 1-based and counts from the beginning of that scriptlet. Optionally, a JavaScript string may be supplied which is evaluated before the breakpoint is being activated. If the string does not evaluate to a non-zero value or true, the breakpoint is ignored. A second optional parameter passes can be used to set a number of times the breakpoint must be passed before it is activated. The special value <code>nextCall</code> , when used as a single parameter, sets a breakpoint on the next function call. This breakpoint is cleared when executed.
<code>clearbp</code> <code>(String</code> <code>scriptlet,</code> <code>Number line)</code>	Clear a breakpoint. The scriptlet parameter is the name of a scriptlet as defined when the scriptlet was entered into the engine. The line number is 1-based and counts from the beginning of that scriptlet. The special value <code>nextCall</code> , when used as a single parameter, clears a previously set breakpoint on the next function call. If <code>clearbp()</code> is used without arguments, all breakpoints are cleared.
<code>bp()</code>	Execute a breakpoint. This method is equivalent to the Javascript debugger statement.
<code>gc()</code>	Call the built-in garbage collector to free up objects and reduce memory requirements.

11

Event-Handling Functions

GoLive calls these functions in response to events such as changes in the state of a control, or parsing the active HTML document. To respond to such events, your extension implements the JavaScript functions this section describes.

Global Functions

initializeModule

```
initializeModule()
```

Called after the module has been loaded. Use it to define global variables etc.

terminateModule

```
terminateModule()
```

Called before the module is being unloaded. Use it to do any final action necessary.

Custom Boxes

parseBox

```
parseBox (box)
```

When GoLive parses a custom tag, it creates a box object, inserts it into the document view, and then calls this function, passing the box as its argument. Your implementation of this function must initialize the box in any way that is appropriate; for example, this function might set the box's height and width as specified by the height and width attributes of the custom HTML element.

IMPORTANT: *Do not call Markup.setInnerHTML(), Markup.setOuterHTML(), Document.reparse() or Document.reformat() from within this function. An attempt to call these methods results in a runtime error.*

GoLive calls the `parseBox` method in response to any of the following events:

- The user drops an Objects palette icon onto a GoLive document window.
- GoLive reads a document containing a custom HTML element defined by the `jsxelement` and `jspxalettentry` elements.
- The user switches to Layout View from another view in the document window.

drawBox

`drawBox (box, draw)`

Called when a custom box is about to be drawn. Implement it to visualize your custom box. Use the global `draw` object to draw lines, rectangles, circles, or text. Use the supplied `Draw` object for the drawing operations.

IMPORTANT: *Your `drawBox` function must not call any functions other than drawing functions. Attempting to reparse the document or to download a file from within the `drawBox` method may cause GoLive to terminate abnormally.*

boxResized

`boxResized (box, width, height)`

Called when the size of the box has changed. Implement it to record the changes in the markup code and/or to recalculate the contents of the box.

inspectBox

`inspectBox (box)`

Called when the inspector dialog for a custom box is about to be displayed. Use it to fill in the control of the inspector with the current values for the box. When an inspector is active for a box, the property `Inspector.box` of the inspector points to the box being inspected, so you can use `control.parent.box` within the `controlSignal()` function to access the box being inspected. On the other hand, when a box is being inspected, the property `Box.inspector` is set to the inspector dialog so you can use the expression `box.inspector.controlName` to access the inspector's controls.

Controls

These functions are called when the state of a control has changed, or when a menu selection has been made.

controlSignal

`controlSignal (control)`

Called when the state of a control has changed. Use it to extract the state of the control and to react on the state change, like setting an URL when the control is an URL getter control.

menuSetup

`menuSetup (menuitem)`

When a menu item has the dynamic attribute set, this function is called when the menu is about to be displayed. This allows the setting of a check mark or to disable the menu item before it is displayed.

menuSignal

`menuSignal (menuitem)`

When a menu item has been selected, this function is called with the affected menu item.

undoSignal

`undoSignal (undo, action)`

Callback function for Undo objects. GoLive calls this function for three purposes, which are reflected in the action parameter:

- 0 – Do. Called as soon as `submit()` is called.
- 1 – Undo.
- 2 – Redo.

IMPORTANT: *Do not call `Document.reparse()` or `Document.reformat()` from within this function. An attempt to call these methods results in a runtime error.*

linkChanged

`linkChanged (link)`

When the user changes a link, either in an URLGetter control or maybe in the Site View, GoLive calls this method with the affected link as parameter.

Custom Controls

For custom controls, GoLive calls the functions this section describes in addition to calling those the previous section describes.

drawControl

`drawControl (control, draw)`

Called for custom controls when they are about to be drawn. Use it to visualize the state of your control. Pay respect to the mouse state (is the control clicked?) when drawing. Use the supplied Draw object for the drawing operations.

IMPORTANT: *Your drawControl function must not call any functions other than drawing functions. Attempting to reparse the document or to download a file from within the drawBox method may cause GoLive to terminate abnormally.*

mouseControl

`mouseControl (control, x, y, mode)`

Called to record mouse clicks on the control. x and y are the position of the mouse pointer relative to the upper left corner of the control. mode is one of the following values:

0 - the left button has been pressed

1 - the mouse has been moved with the left button down

2 - the mouse button has been released

When this function is not present, controlSignal() gets called instead when the mouse button is released over a custom control.

12

C and C++ APIs For Use In External Binary Libraries

This chapter describes C-language macro functions that the `JSA.h` interface (header) file provides. The functions in a compiled external library call these functions to

- obtain arguments from the GoLive environment
- pass return values back to GoLive.

For a guide to the use of these functions, see [Appendix A, “Using External Libraries.”](#)

C API Synopsis

This section summarizes use of the C API. For more detail, see [“Implementing External Binary Libraries” on page 187](#).

```
// include JSA header for C or C++
#include "JSA.h" // use "JSA++.h" for C++ implementations

// include any other libs your extension requires
#include <math.h>

// include platform-specific drawing support
#ifdef WIN32
    #include <windows.h>
#else
    #include <QuickDraw.h>
#endif

// Call this macro once to initialize the JavaScript environment.
JSA_INIT

// define your functions
// use argc/argv pairs to pass args
// use returnValue to return this fn's result
static void power(int argc, JSValue *argv, JSValue returnValue)
{
    double a, b, c;
    // call JSAXxx fns to extract args passed by JavaScript callers
    a = JSAValueToDouble(argv[0]);
    b = JSAValueToDouble(argv[1]);

    c = pow (a, b);

    // call JSAXxx fns to pass values back to JavaScript callers
    JSADoubleToValue(returnValue, c);
}

// you must implement this fn, which registers your lib's fns with GoLive
void JSAMain() {
    JSRegisterFunction ("power", power);
}

/*****
/* Assuming you've built this fn into a library named JSASample.dll (or */
/* just JSASample for Mac OS platforms) which resides in the          */
/* "Modules/Extend Scripts/Common" folder, your JavaScript code calls */
/* the external library function as follows:                            */
*****/
JSASample.power (2,3);
```

C++ API Synopsis

The C++ API is a superset of the C API. A C++ library is written, built, and used in exactly the same manner as a C library. However, library functions written in C++ have direct access to the C++ objects that encapsulate JavaScript arguments and return values; thus, they can manipulate these objects directly or they can use the JSAXxx functions just as a C library would.

The following code example illustrates the C++ implementation of the power function shown in the preceding section.

```
static void power(int argc, const sValue* argv[], sValue* returnValue)
{
    double a = argv[0]->getDouble();
    double b = argv[1]->getDouble();

    double c = pow (a, b);

    returnValue->setDouble (c);
}
```

Data Types

This section describes the data types GoLive provides for use by external binaries.

JSValue pointer

The JSValue type is an opaque void pointer that is a data element in the *argv* vector GoLive passes to an externally-defined binary function. Internally, GoLive casts this pointer's type as necessary to hold each element's data.

```
typedef void *JSValue;
```

JSValueType Scalar Types

The following macros define JSValueType scalar data types as returned by the JSGetValueType function.

Undefined

```
JSA_UNDEFINED
```

The undefined or empty value.

Boolean

JSA_BOOL

A Boolean value, expressed as an integer. 0 represents a value of false, and 1 represents a value of true.

Integer

JSA_INTEGER

A 32-bit signed integer quantity.

Floating Point

JSA_DOUBLE

An 8-byte, double-precision, floating-point value.

String

JSA_STRING

A null-terminated ASCII string.

JSANativeMethod Type

The GoLive external binary API encodes your library's function definitions as JSANativeMethod structures. This data type implements the following data structures, which your functions use to implement their parameters.

<i>argc</i>	<i>Integer</i>	Number of array elements in the <i>*argv</i> vector.
<i>*argv</i>	<i>JSValue</i>	Vector of argument values
<i>return</i>	<i>JSValue</i>	Empty pointer passed to your function by GoLive when it is called. C functions call the appropriate JSAXxToValue function to store a return value in this pointer. C++ functions can access this pointer directly or they can call a JSAXxToValue function.

JSADrawInfo Struct

The JSADrawInfo struct is used to implement C or C++ drawing functions that can be called from JavaScript. For more information, see [“Drawing Function Examples” on page 190](#).

```
typedef struct _JSADrawInfo {  
    long context; // a DC (Windows) or a GrafPort (Mac)  
    long left, top; // upper left corner of the drawing rect  
    long right, bottom; // lower right corner of the drawing rect  
} JSADrawInfo;
```

Initialization and Termination Functions

This section describes functions used to initialize or terminate an external binary library.

JSA_INIT Macro

`JSA_INIT`

The `JSA_INIT` macro must appear exactly once in the implementation of a binary extension . This macro call must appear after the `JSA.h` or `JSA++.h` include statement and before the required call to the `JSAMain` function.

The `JSA_INIT` macro inlines the `JSAEntry` function. GoLive tests for the presence of the `JSAEntry` function to determine whether an external binary is intended for use by Extend Script extensions; thus, if the external binary does not call the `JSA_INIT` macro, GoLive does not make the external binary available to Extend Script extensions.

The `JSAEntry` function

- sets an environment pointer containing references to the various `JSAXx` functions this Appendix describes.
- calls the `JSAMain` function.

JSAMain

`JSAMain()`

Every external binary must implement the `JSAMain` function.

Your implementation of this function must register your external binary's functions with the GoLive JavaScript engine. To do so, it calls the `JSRegisterFunction` function once for each function it registers. Optionally, your implementation of the `JSAMain` function can perform any additional initialization tasks your external binary requires.

JSARegisterFunction

```
JSARegisterFunction(name, foo)
```

Register the function *foo* under the name specified by the value of the *name* parameter.

The JSARegisterFunction function makes an external binary function available under a specified JavaScript name. Only registered JSANativeMethod functions are available to Extend Script extension modules.

To register a function, call the JSARegisterFunction function from within the body of the external library's JSAMain function. You must call JSARegisterFunction function once for each function to be registered.

Arguments

<i>name</i>	<i>String</i>	The name under which the foo function is to appear in the JavaScript namespace. This name must observe JavaScript naming conventions.
<i>foo</i>	<i>Token</i>	The token that the function definition associates with the function's implementation. For example, the code <pre>myFunk () {return;}</pre> associates the myFunk token with the {return;} implementation.
<i>return</i>	<i>Integer</i>	Optional. An integer value to return instead of the result the call to this function actually returns.

JSASExit

```
JSASExit()
```

GoLive calls your external binary's JSASExit function when the extension module that uses it about to be unloaded. Your implementation of this optional method perform housekeeping tasks required to exit the extension, such as setting the values of your pointers and variables to null.

Arguments

None.

Returns

Void.

Accessor Functions

C-language library functions must use these accessors to extract arguments from GoLive and return values to GoLive. C++ functions can access the C++ objects in the GoLive engine directly, or they can use these accessors in the same way that C-language functions do.

JSGetValueType

`JSGetValueType(arg)` returns an integer value that indicates the type of the *arg* argument.

Synopsis

```
#include "JSA.h"
. . .
JSGetValueType(arg, return)
```

Arguments

<i>arg</i>	<i>JSAValue</i>	The JSA value to test
<i>return</i>	<i>Integer</i>	Optional. An integer value to return instead of the result the call to this function actually returns.

Returns

One of the following integer values indicating the type of the JSA object passed as the value of the *arg* parameter:

0	Undefined
1	Boolean
2	Integer
3	Double (double-precision floating point)
4	Text

Example

```
static void myFn(int argc, JSValue *argv, JSValue returnValue)
{
    int a, b, c;
    a = JSGetValueType(argv[0]);
    JSIntToValue(returnValue, a);
}
void JSAMain() {
    JSRegisterFunction ("myFn", myFn);
}
```

JSAValueToInt

`JSAValueToInt(arg)` returns the value of the *arg* parameter as a 32-bit integer.

Synopsis

```
#include "JSA.h"
. . .
JSAValueToInt(arg)
```

Arguments

<i>arg</i>	<i>JSAValue</i>	The JSA value to test
<i>return</i>	<i>Integer</i>	Optional. An integer value to return instead of the result the call to this function actually returns.

Returns

32-bit Integer

JSAValueToBool

`JSAValueToBool(arg)`

Return the argument as a boolean (an integer, either zero or nonzero).

Arguments

<i>arg</i>	<i>JSAValue</i>	The JSA value to test
<i>return</i>	<i>Integer</i>	Optional. An integer value to return instead of the result the call to this function actually returns.

JSAValueToString

JSAValueToString(*arg*)

Return the argument as a zero-terminated ASCII string.

Arguments

<i>arg</i>	<i>JSAValue</i>	The JSA value to test
<i>return</i>	<i>Integer</i>	Optional. An integer value to return instead of the result the call to this function actually returns.

JSAValueToDouble

JSAValueToDouble(*arg*)

Return the argument as an eight-byte floating point value.

Arguments

<i>arg</i>	<i>JSAValue</i>	The JSA value to test
<i>return</i>	<i>Integer</i>	Optional. An integer value to return instead of the result the call to this function actually returns.

JSAIntToValue

JSAIntToValue(*arg*, *val*)

Store the long value *val* into *arg*.

Arguments

<i>arg</i>	<i>JSAValue</i>	The JSAValue object that is to hold the <i>val</i> argument.
<i>val</i>	<i>Integer</i>	32-bit integer value to store in the <i>arg</i> object.

JSABoolToValue

```
JSABoolToValue(arg, val)
```

Store the integer value *val* into *arg*. Non-zero values of *val* specify a Boolean value of true.

Arguments

<i>arg</i>	<i>JSValue</i>	The JSValue object that is to hold the <i>val</i> argument.
<i>val</i>	<i>Integer</i>	32-bit integer value to store in the <i>arg</i> object. Non-zero values of <i>val</i> specify a Boolean value of true.

JSAStrToValue

```
JSAStrToValue(arg, val)
```

Store the value stored in the zero-terminated ASCII string variable pointed to by *val* into *arg*.

Arguments

<i>arg</i>	<i>JSValue</i>	The JSValue object that is to hold the <i>val</i> argument.
<i>val</i>	<i>String</i>	Zero-terminated ASCII string to store in the <i>arg</i> object.

JSADoubleToValue

```
JSADoubleToValue(arg, val)
```

Store the double value *val* into *arg*.

Arguments

<i>arg</i>	<i>JSValue</i>	The JSValue object that is to hold the <i>val</i> argument.
<i>val</i>		Double-precision floating-point value to store in the <i>arg</i> object.

JSUndefinedToValue

`JSUndefinedToValue(arg)`

Set the value of *arg* to undefined.

Arguments

<i>arg</i>	<i>JSValue</i>	The <i>JSValue</i> object that is to hold the <i>val</i> argument.
------------	----------------	--

JSSetError

`JSSetError(text)`

Generate a Javascript runtime error with the given text as explanation.

Arguments

<i>text</i>	<i>String</i>	The text of the error message this function generates.
-------------	---------------	--

JSSEval

`JSSEval(text, timeout)`

Evaluate a specified Javascript scriptlet in the current execution context (inside the function which called the native code function).

Arguments

<i>text</i>	<i>String</i>	The expression the JavaScript engine is to interpret.
<i>timeout</i>	<i>Integer</i>	Positive values specify the number of milliseconds to wait for the call to complete. If the timeout elapses, the engine generates a runtime error. A value of 0 causes the engine to run the scriptlet asynchronously; that is, the call returns immediately, regardless of whether the scriptlet completes execution successfully. A <i>timeout</i> value that is less than zero causes the caller to wait unconditionally.

Returns

The return value of this call is a *JSValue* pointer that points to the result of the scriptlet. If no timeout was set or the timeout elapsed, this value is *undefined*.



Example

External library functions use the `JSAEval()` function to retrieve the values of JavaScript variables; for example, the following line of code returns the value of the `myVariable` global variable.

```
JSAEval ("myVariable", -1)
```

Draft

Part III

Appendixes

Draft



Draft

A

Using External Libraries

Extend Script extensions can call JavaScript, C, and C++ functions provided by one or more optional external libraries. This Appendix describes how to create and use external libraries.

Benefits of External Libraries

The use of an external library is entirely optional. Most Extend Script extensions use only JavaScript and the special tags that the SDK provides.

This book refers to an external library written in JavaScript as a **script library**. An external library written in C or C++ must be compiled specifically for the Mac OS or Windows platform on which it is to run. This book refers to a compiled library as an **external binary library**.

The only time an extension actually requires the use of an external binary is when it must perform tasks for which JavaScript is not suitable. However, extensions often use external libraries for other benefits they offer, such as convenience, performance improvements, data-hiding features, and the cross-platform deployment of compiled code,

- **Convenience**

One of the best reasons to use an external library is for the convenience it provides to the extension developer. For example, if you have a common set of functions you'd like to use in multiple extensions, you could package them in an external library that all of the extensions can call, rather than duplicating them in the source code of each extension that uses them.

If you already program in C or C++, you might find it easier to use these languages instead of JavaScript, in some cases. External libraries provide the means by which you can do so.

- **Reduced Code Size**

Using external libraries can reduce an extension's source code size, but it does not reduce the extension's memory requirements.

- **Performance Improvements From Compiled Binary Libraries**

Because compiled code runs faster than interpreted code, you may realize performance improvements by implementing some functions in C or C++ rather than in JavaScript.

- **Data Hiding in Compiled Binary Libraries**

JavaScript provides no data-hiding capabilities. To hide the implementation of proprietary code, you can implement it in C or C++ as a compiled external library. External JavaScript libraries offer no data-hiding capabilities.

- **Cross-Platform Deployment of Compiled Code**

External libraries enable you to deploy C or C++ functions to GoLive extensions running on Mac OS and Windows platforms.

External JavaScript Libraries

As the name implies, a JavaScript library is written in JavaScript. You can use GoLive or any text editor to create a JavaScript library.

Implementing A JavaScript Library

The code that implements a JavaScript function in an external JavaScript library is no different than the code that implements the same function from within an extension's `Main.html` file—the only difference is its location, which is a file that

- has a `.js` file extension.
- is located in the `Extend Scripts/Common` folder.

NOTE: Don't confuse `.js` files with the `.sct` files used by previous versions of GoLive. **Extend Script** extensions cannot call `.sct` files.

Installing An External JavaScript Library

To make an external library available to Extend Script, place its `.js` file in the **Common** subfolder inside the **Extend Scripts** folder.

IMPORTANT: *GoLive loads only the libraries located in the **Common** folder. Libraries not located in the **Common** folder are not available to Extend Script extensions.*

When GoLive starts, it loads all of the `.js` files in the **Common** subfolder, as well as any JavaScript code defined by `Main.html` files in the **Extend Scripts** folder. Thus, functions defined “internally” by the `Main.html` file execute no more quickly than those defined “externally” by a `.js` library file.

External JavaScript libraries aren't treated exactly like the JavaScript in the `Main.html` file, however. An external library is meant to define functions only—it should not contain any other code.

NOTE: Only the function definitions of a script library file are added to each module when GoLive starts. When loading an external script library, GoLive ignores all other code the library defines; for example, your script library should not define global variables, tags, or elements—such code does not even execute.

Calling JavaScript Library Functions

GoLive makes external JavaScript library functions available to the scripts of an extension as if they were part of the extension itself. Using an external JavaScript libraries does not reduce memory requirements—every module loads its own copy of the external functions when GoLive starts.

Implementing External Binary Libraries

This section provides detailed instructions for implementing and building external binaries in the C and C++ programming languages.

An external C or C++ library is implemented as a binary file:

- On Windows platforms, the library is a dynamically-linked library (DLL).
- On Mac OS platforms, the library is a shared library.

You can build these binaries using any C or C++ compiler that can build a shared library for Mac OS platforms or a DLL (dynamically-linked library) for Windows platforms. The SDK provides project files for Microsoft® Visual C++ 6.0 and Metrowerks CodeWarrior 5 Pro.

Including C Libraries

The source files that define an external binary library must include files that provide

- the types and functions this Appendix describes
- platform-specific support
- other libraries or definitions on which the external binary is dependent

This section describes the following `#include` statements, which provide these resources.

```
// include JSA header for C or C++
#include "JSA.h" // use "JSA++.h" for C++ implementations

// include any other libs your extension requires
#include <math.h>

// include platform-specific drawing support
#ifdef WIN32
    #include <windows.h>
#else
    #include <QuickDraw.h>
#endif
```

Including JSA Interface Files

To use the types and functions this Appendix describes, your external library must include the `JSA.h` or `JSA++.h` interface file:

- An external library implemented in C must `#include` the `JSA.h` file, as the following example does.

```
#include "JSA.h" // use "JSA++.h" for C++ implementations
```

- An external library implemented in C++ must `#include` the `JSA++.h` file to access C++ objects in the GoLive environment directly.

```
#include "JSA++.h"
```

If the library does not access these objects directly, but uses only JSAXx accessor functions to interact with them, it can `#include` either of the `JSA.h` or `JSA++.h` files.

Including Platform Support

The `Windows.h` and `QuickDraw.h` files provide platform-specific support for Windows and Mac OS host systems, respectively. The external binary's sources must include the interface file that is appropriate for the platform on which the external binary is to be used; to do so, you can use the `WIN32` environment variable in a conditional include statement like the one in the following example.

```
#ifndef WIN32
    #include <windows.h>
#else
    #include <QuickDraw.h>
#endif
```

Including Additional Interface Files

Your external binary's sources should also include any other files that provide functionality on which it depends. For example, the `JSASample.c` implementation file uses the `pow` function that the `math.h` interface file provides, so it includes this interface file as follows:

```
#include <math.h>
```

Initializing the JavaScript Engine

To ensure that the module is intended for access through Javascript, GoLive checks for the presence of the `JSAEntry` function. This function is defined by the `JSA_INIT` macro, which must appear once in your extension's code.

The `JSAEntry` function sets an environment pointer containing a number of function references, and then it calls the `JSAMain` function. Your implementation of the `JSAMain` function registers your external library functions with GoLive and performs any initialization tasks your extension requires.

```
// Call this macro once to set up the necessary structures.
```

```
JSA_INIT
```

Defining External Library Functions

When defining your external library's functions, use the `JSValue` pointer type to exchange data with the JavaScript environment.

Define your library's functions using the following keywords and types to define a parameter block containing an array of arguments and an optional return value:

`argc` number of arguments in the `argv` array

`argv` array of `JSValue` pointers to pass as arguments to this function

`result` a `JSValue` pointer used for temporary storage of the value that this function returns

Math Function Example: C Language

When an extension calls an external library function, GoLive encapsulates the arguments passed from JavaScript as C++ objects. If the called function is implemented in C, it must use the [“Accessor Functions” on page 177](#) to extract its arguments from the C++ objects that encapsulate them.

A typical C-language implementation would look like the following:

```
#include "JSA.h"
#include <math.h>
JSA_INIT;
// Return arg1arg2
static void power(int argc, JSValue *argv, JSValue returnValue)
{
    double a, b, c;
    a = JSValueToDouble(argv[0]);
    b = JSValueToDouble(argv[1]);

    c = pow (a, b);

    JSADoubleToValue(returnValue, c);
}

void JSAMain() {
    JSRegisterFunction ("power", power);
}
```

NOTE: Do not attempt to replace a standard library function by defining your own version of it in an external C library.

Math Function Example: C ++

In contrast to the C API, the C++ API provides full access to the runtime engine and the C++ objects that encapsulate JavaScript arguments; thus, C++ library functions can manipulate these values directly or they can use the accessor functions that C-language functions must use.

```
static void power(int argc, const sValue* argv[], sValue* returnValue)
{
    double a = argv[0]->getDouble();
    double b = argv[1]->getDouble();

    double c = pow (a, b);

    returnValue->setDouble (c);
}
```

Drawing Function Examples

It might often be desirable to implement the drawing of a custom box in native code. The GoLive Extend Script SDK supports this with the `getDrawInfo()` method of the `Draw` object. This method returns a magic integer number which can be passed on to a native code function. This function can cast the magic number to a pointer to a `JSADrawInfo` structure which contains all necessary drawing information:

```
typedef struct _JSADrawInfo {
    long context; // a DC (Windows) or a GrafPort (Mac)
    long left, top; // upper left corner of the drawing rect
    long right, bottom; // lower right corner of the drawing rect
} JSADrawInfo;
```

The drawing rect describes the entire rectangle of the box to draw. This rectangle may be clipped if only parts of the rectangle are to be drawn. In any case, the drawing context is clipped to the bounds of that rectangle. The drawing context is an offscreen drawing context pointing to a bitmap, where certain restrictions might apply). Please make sure that any temporary changes to the drawing context are undone before returning!

An external C function uses the `JSADrawInfo` structure as the following example does:

```

// Draw an oval into the given rectangle. This demonstrates the
// use of the JSADrawInfo structure.

static void drawOval(int argc, JSValue *argv, JSValue returnValue)
{
    long temp;
    JSADrawInfo* info;
#ifdef WIN32
    HDC dc;
    temp = JSValueToInt (argv[0]);
    info = (JSADrawInfo*) temp;
    dc = (HDC) info->context;
    Ellipse (dc, info->left, info->top, info->right, info->bottom);
#else
    Rect r;
    temp = JSValueToInt (argv[0]);
    info = (JSADrawInfo*) temp;
    r.left = (short) info->left;
    r.top = (short) info->top;
    r.right = (short) info->right;
    r.bottom = (short) info->bottom;
    FrameOval (&r);
#endif
}

```

Because it has direct access to the C++ objects that encapsulate arguments from JavaScript callers, the C++ version of the drawOval function can provide this functionality in a slightly different way:

```

// Draw an oval into the given rectangle. This demonstrates the
// use of the JSADrawInfo structure.

static void drawOval(int argc, const sValue* argv[], sValue* returnValue)
{
    int temp = argv[0]->getInteger();
    JSADrawInfo* info;
#ifdef WIN32
    HDC dc;
    info = (JSADrawInfo*) temp;
    dc = (HDC) info->context;
    ::Ellipse (dc, info->left, info->top, info->right, info->bottom);
#else
    info = (JSADrawInfo*) temp;
    Rect r;
    r.left = (short) info->left;
    r.top = (short) info->top;
    r.right = (short) info->right;
    r.bottom = (short) info->bottom;
    ::FrameOval (&r);
#endif
}

```

The language used to implement an external function has no effect on how it is called from JavaScript. Assuming that either version of the `drawOval` function is built into an external library named `JSASample`, your JavaScript code calls the function as follows:

```
function drawControl(control,draw) {
    if (control.name == "custom")
        JSASample.drawOval (draw.getDrawInfo());
}
```

Registering External Functions

All external binaries must implement the `JSAMain` function as described here. Your implementation of this function must register each of your external library functions with GoLive. To do so, it passes each function's JavaScript and C-language names to the `JSARegisterFunction` function, as in the following example.

```
void JSAEXPORT JSAMain(void)
{
    JSARegisterFunction("power",power);
    JSARegisterFunction("drawOval",drawOval);
}
```

The first argument is the name under which the function is to appear in JavaScript. The second is the name of the function's C or C++ implementation. Both functions this example registers happen to have identical names in JavaScript and in C or C++, but these names do not need to match.

When the `JSAMain` function completes, all of the functions it registered can be called from JavaScript by any Extend Script extension. The functions are made accessible as the properties of an object named for the binary module; thus, if both of the functions in the preceding example were built into a library named `JSASample.dll` (or just `JSASample` on Mac OS) they could be called from JavaScript as follows:

```
result = JSASample.power(2,4)

function drawControl(control,draw) {
    if (control.name == "custom")
        JSASample.drawOval (draw.getDrawInfo());
}
```

Implementing Optional Termination Code

When a module is unloaded, GoLive calls the function `JSAExit()`, which can be implemented to contain any cleanup code.

Building An External Binary Library

You can use any Mac OS or Windows C/C++ compiler to build the library as a shared library for Mac OS platforms or as a DLL for Windows platforms.

Since there are structures involved with this SDK, correct structure alignment settings are crucial. For the Macintosh, structure alignment is 68K, and for Windows, structure alignment is 8 bytes.

Installing An External Binary

To make an external library available to Extend Script, place it in the **Common** subfolder inside the **Extend Scripts** folder. This folder can hold any number of external libraries. The functions defined by these libraries are available to all Extend Script extension modules.

IMPORTANT: *GoLive loads only the libraries located in the **Common** folder at start time. Libraries not located in the **Common** folder when GoLive starts are not available to Extend Script extensions.*

GoLive loads external binary functions and JavaScript functions differently. All JavaScript functions are loaded in memory immediately at startup time; in contrast, binary libraries are registered with GoLive but not loaded until an extension calls a function that the library provides. The time required to load the library makes the first call to an external binary library a little slower than subsequent calls to that library.

Calling C and C++ Library Functions From JavaScript

The functions an external binary provides can be called by any extension. The external library is loaded on demand the first time it is called; subsequently, it remains in memory until the module is unloaded. The module is unloaded

- when GoLive quits
- when the user chooses the **Reload Scripts** palette item
- when the `moduleName.unload()` function is called.

NOTE: The `unload` function destroys its module's global data. The next time a function in the library is called, GoLive reloads the module and creates new global data for it.

JavaScript code calls external C and C++ library functions identically. If we assume that the above code is part of the file `Utils.dll` (Windows) or `Utils` (Mac OS), this function would be called from Javascript as follows:

```
a = Utils.power (2,5);
```

Measuring Performance

One useful metric for testing code performance is the time required to complete a function call. You can use the `Date` object provided by JavaScript to measure code performance by getting the system time in milliseconds immediately before and after calling the function being tested. For example, the code to test the execution speed of scripted and compiled versions of the same function would look like the following example.

```
// measure time for script fn to complete
date1 = new Date()
myScriptFunction()
date2 = new Date
scriptTime= (date2-date1)
writeln("Script code execution time" + scriptTime + "milliseconds")

// assume the compiled fn is built into myExtLib.dll
// measure time for compiled fn to complete
date3= new Date()
myExtLib.myCompiledFunction();
date4 = new Date
compiledTime = (date4-date3)
writeln("Compiled code execution time" + compiledTime + "milliseconds")
diff = (scriptTime - compiledTime)

if (diff == 0)
    writeln ("Both functions executed in the same amount of time.")
if (diff > 0)
    writeln("Compiled code was faster by "+ diff + "milliseconds.")
else
    writeln("Script code was faster by" + (-diff) "milliseconds")
```

B

Sort Order Tables

This chapter lists the values GoLive uses to sort items in the Windows menu and the Objects palette.

Window Menu Items

The entries of the Window menu are sorted groupwise. The upper two digits denote a group, while the lower two digits are the sort order within the group. Groups are separated by menu separators, which means that f.ex. all order numbers which start with 90xx are grouped together and separated from the other groups with a menu separator. [Table B.1](#) lists the values used to sort Window menu items.

TABLE B.1 *Codes used to sort Window menu items*

Objects	0101
Color	0110
Inspector	2001
View Controller	2001
Debug	3001
Transform	3001
Align	3002
Tracing Image	3003
Floating Boxes	5001
Table	5002
Link View	7001
Source Code	7002
Javascript Shell	7003
Markup Tree	9001
History	9003

Objects Palette Entries

The objects in the **Objects** palette are sorted in two levels. The first level is the order the tabs for each group appear in, while the second value is the sort order within one group of icons. The higher the value is, the further to the right the icon or tab appears. Please note that many built-in modules contain more than one icon. This is for example the case for all icons within the Head section. New icons cannot be placed between these icons, only left or right of these icons. [Table B.1](#) lists the values used to sort **Objects** palette entries.

TABLE B.2 Codes used to sort Objects palette entries

Name	taborder	order	icons
Basic	0		
Layout Grid		10	2
Floating Box		15	1
Table		20	1
Image		30	1
Plugin		40	5
Java™ Applet™		50	1
Object		60	1
Line		70	1
Horizontal Spacer		80	1
JavaScript		90	1
Marquee		100	1
Comment		110	1
Anchor		120	1
Line Break		130	1
Tag		140	1
(reserved for future use)		150	1
Smart	1		
Smart items		200	all
Forms	2		
Form items		10	all
Head	3		
Head items		20000	all
Frames	4		

Frame items		20000	all
WebObjects	5		
WebObjects items		1-4	all
Site	50		
Site items		1	all
Site Extras	51		
Site extras items		2	all
QuickTime	5000		
QuickTime items		1	all
Custom	20000		
Custom items		20000	all
Extensions (default)	30000		
Extension items (default)		30000	



Draft



Glossary

Element

A [Tag](#) and all of the attribute values required to define an instance of the entity the tag represents.

Parse

To read the elements in a document and generate a tree of markup objects representing the elements the document defines. GoLive parses the active document in response to any of the following events:

- The user drops an Objects palette icon onto a GoLive document window.
- GoLive reads a document containing a custom HTML element defined by the `jsxelement` and `jsxpalettentry` elements.
- The user switches to Layout View from another view in the document window.

Tag

Alphanumeric tokens enclosed by angle brackets (<>), as in the ``, ``, or `<H1>` tags. Tags that are used singly, such as the `` tag, are **unary tags**. Tags that must be used in pairs are **binary tags**; for example, the `<H1>` opening tag must always be paired with an `</H1>` closing tag. When this book refers to a binary tag, it names the opening tag only and assumes you understand that the presence of the closing tag is implied.



Draft

Index

A

- alert dialog
 - displaying 44
- alert function 44
- Alt key 46
- attributes
 - dynamic 49
 - key 45
 - name 35

C

- checked property 48
- Cmd key 46
- Ctrl key 46

D

- drawBox function 33
- drawControl function 33
- drawing functions
 - errors in 33
- dynamic attribute 49

E

- enabled property 48
- Encodings module 29
- event-handling functions
 - menuSetup 50
 - menuSignal 44
- Extend Script extension
 - installing 27
- Extend Script module
 - enabling 26

F

- functions
 - alert 44

- drawBox 33
- drawControl 33
- menuSetup 50
- menuSignal 44

G

- global objects
 - menus 35

J

- Japanese character sets
 - Encodings module and 29
- JavaScript
 - version information, flavor information 17
- jsxcontrol
 - characteristics of 55
- jsxitem
 - checked property of 48
- jsxitem tag 43
 - dynamic attribute of 49
- jsxmenu tag 42
- jsxmenubar tag 42
- jsxmodule tag 33

K

- key attribute 45
- keyboard shortcuts
 - assigning to menu items 45
- keyboard shortcuts and 45

L

- library
 - C language 17
 - C++ language 17

M

menu

- adding to menu bar 42
- checked state of 48
- defining name of 42
- initializing 50
- overview of creating 42

menu bar

- adding menu to 42

menu items

- assigning keyboard shortcuts to 45
- defining 43
- disabling 48
- enabling 48
- initializing dynamically 49
- Special menu and 51

menus array 35

menuSetup function 50

menuSignal function 44

modifier keys 46

modules

- creating 38
- disabling 29
- Encodings 29
- Network 29

N

name attribute 35

name property 35

- object comparison and 36

Network module 29

O

Option key 46

P

properties

- enabled 48
- name 35

R

reparsing

defined 34

S

scripts

- execution timeout 33

shared libraries 17

Special menu

- adding items to 51

submenu

- defining 47

T

tags

- jsxitem 43
- jsxmenu 42
- jsxmenubar 42
- jsxmodule 33

timeout

- script execution 33

U

Unicode 17

user alert

- displaying 44