APPENDIX E

Extend Script and the GoLive SDK

One of the more complex and powerful new features introduced with GoLive 5 is Extend Script. Extend Script enables you to add new capabilities and features to GoLive by writing code in JavaScript and the special markup tags provided by the Extend Script SDK (Software Development Kit). You can add new menus, palettes, custom boxes, inspectors, and more. It is beyond the scope of this book to examine Extend Script in great detail but this appendix briefly outlines how to go about writing a basic GoLive Extension. It is assumed that you have a basic understanding of JavaScript and HTML.

It is also possible to write binary extension modules using C/C++ instead of JavaScript. This enables you to add very sophisticated capabilities to GoLive beyond what is possible with standard Extend Script. Writing such code requires knowledge of C compilers and is for advanced developers only.

Getting Started with Extend Script

To use extension modules created with the SDK and to test your own extension modules, you must first ensure that you have the Extend Script module enabled in preferences. The GoLive installer activates this module by default, so you should not have to make any changes unless you have disabled it.

You may want to first try out some of the sample extensions that come with the SDK to give you an idea of what it can do. If you have a Mac, you will find the SDK samples in the SDK folder on your GoLive installation CD-ROM. If you use

Windows, the samples are installed on your hard disk into the same folder as the GoLive application. To install the sample extensions, simply drag them into the Extend Scripts subfolder of the Modules folder.

The SDK documentation and sample code are regularly updated. By the time this book is published there will probably be a newer version of the SDK available from the Adobe Web site at `http://partners.adobe.com/asn/developer/gapsdk/GoLiveSDK.html`. Always make sure you are using the latest version of the SDK.

Examining the Structure of an Extend Script

Every Extend Script module is defined by a file called Main.html. Each script has its own Main.html file that is located in its own folder in the Extend Scripts folder. If any images or other resources are required by the module (for instance if the Extend Script defines a new palette object or a custom box), they must be located in the same folder as the Main.html file. You can see the directory structure of an Extend Script in Figure E-1.

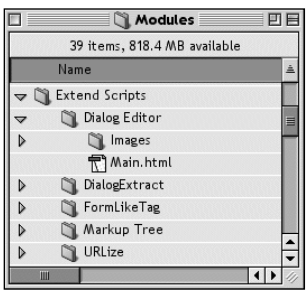

Figure E-1: The Extend Scripts folder structure

Although the Main.html file is a normal text file and contains the `<head>`, `<body>`, and other HTML tags, you never actually view this page in a browser. The Main.html file contains special SDK-specific markup tags and JavaScript code that GoLive interprets when it is launched and the code becomes part of the GoLive application.

Because GoLive loads Extend Script modules at launch, testing your modules can be slow because you have to relaunch GoLive every time you make changes to the code. You can reduce the time it takes to launch GoLive whilst testing by using the preferences dialog box to disable modules you aren't currently using.

The basic layout of a GoLive SDK extension is as follows:

```
<html>
<head>
<title>GoLive 5 Extension Module</title>
</head>
<body>
  <!--Module name-->
<jsxmodule timeout=0 debug>

<script language="javascript"><!--
    //JavaScript that processes events goes here
    //-->
    </script>

    <!--various SDK tags to define palettes, menus etc. go
here-->
  </body>
</html>
```

As you can see, it does not look very different to a standard HTML page at this point.

The GoLive SDK provides several special markup tags that you use to define new interface elements in GoLive such as palettes, menus, inspectors, custom tags, and dialog boxes. All the GoLive SDK tags start with the letters *jsx* to distinguish them from other tags. Table E-1 provides a list of the all the SDK tags and an explanation of what they do.

You are free to mix normal HTML tags with the special SDK tags in your Main.html page. Most of these will be ignored by GoLive unless they are contained within SDK tags, but they can be used as a way to display usage and copyright information should a user directly open your Main.html file in GoLive.

As with normal HTML tags, these special tags each have their own set of attributes that modify their behavior. For instance, you can set the width and height of a fixed-size custom box by defining the `fixedWidth` and `fixedHeight` attributes of the `<jsxelement>` tag.

You don't need to write the tags in any special order in the Main.html file, but some tags require other tags to be present. An example of this is the `<jsxinspector>` tag; because it presents an inspector for a custom box, you must define the custom box as well for it to function properly.

Table E-1
GoLive's SDK Tags and Their Functions

Tag	*Function*
<jsxmodule>	Defines the name of the module, the error timeout setting, and the debug status
<jsxlocale>	Enables you to supply internationally localized versions of the text presented by your module
<jsxdialog>	Defines a dialog box
<jsxpalette>	Defines a floating palette similar to the Inspector window
<jsxcontrol>	Defines controls such as checkboxes, edit fields, and URL fields for palettes and inspectors
<jsxpalettegroup>	Defines a "tab" in the Object palette that holds palette items
<jsxpaletteentry>	Defines an individual palette item and the custom tag or markup that is written to the page when the palette item is used
<jsxelement>	Defines a custom box
<jsxinspector>	Defines an inspector
<jsxmenubar>	Surrounds a block of menu-definition statements
<jsxmenu>	Defines an individual menu or submenu
<jsxitem>	Defines a menu item
<img>	Used to reference any images used by your extension, typically palette icons

Exploring JavaScript and the GoLive Object Model

Once you have defined the visual elements of the extension using markup tags, you can actually set about making your extension do something useful. You bring your extension to life using JavaScript.

GoLive 5 exposes the markup tree of a document as a standard JavaScript object model. This means that all the markup elements of a page are accessible as JavaScript Objects, and the attributes of the markup elements are accessible as properties of those objects. You can also reference several global objects and properties, such as the name of the current module.

The JavaScript Shell

GoLive 5 provides you with a JavaScript Shell palette that is extremely useful when developing extension modules, as it enables you to view the current properties of any object simply by typing the name of the object property into the command field. If the JavaScript shell is not visible on your screen, you can make it visible by selecting it from the Window menu. Try testing the JavaScript shell by typing **document.title** into the command field of the shell and pressing Return or Enter. You should see the name of the currently open document appear in the results pane, as shown in Figure E-2. If no document is currently open, the result will be an error. You can also directly execute JavaScript commands using the shell. Try typing **alert("Hello, world");** into the command field and pressing Return or Enter.

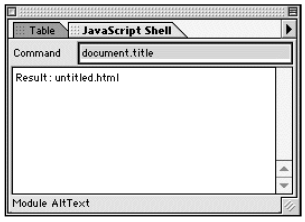

Figure E-2: The JavaScript Shell palette

As stated previously, if you make changes to the code of your extension module, you need to quit and restart for the changes to take effect. However, if you are making changes only to the JavaScript and not to the markup tags, you don't need to quit GoLive—you can reload the JavaScript for all your extensions on the fly. To do this, use the fly-out menu on the JavaScript Shell palette and select Reload all Scripts. This can save a lot of time.

Functions

All interaction with the elements in your extension is handled by JavaScript functions, which respond to JavaScript events. Various standard functions are called by GoLive when certain events occur—for instance when a user selects a menu item. For instance, GoLive calls the `menuSignal()` function whenever a menu item is selected. If you define the `menuSignal()` function in your extension module, then GoLive runs your code whenever a menu item is selected. Too many standard functions exist to list here, but they are covered in detail in the PDF documentation for the GoLive SDK.

As well as writing code to handle the standard functions, you can define your own functions, just as you can using standard JavaScript running in a browser. In fact, if you are experienced in JavaScript development you may already have some standard functions that can be reused in a GoLive extension.

Writing an Extend Script

To give you a clearer picture of how it all fits together, I'll go through the process of writing a simple Extend Script. This example Extend Script is not particularly useful — it reverses the currently selected text — but it demonstrates nicely how the SDK works.

First create a folder inside the Modules/Extend Scripts folder and give it a unique name. Then use GoLive to create a blank HTML page and save it in the folder you just created using the name Main.html.

The result of this exercise is on the CD-ROM that comes with this book. After you complete your first foray into Extend Script, you can check your work against it. You can find the Script in the Exercises folder, which is organized by chapter.

Defining the module

Switch to Source view. The first tag you will add is the `<jsxmodule>` tag. Type the following after the initial `<body>` tag:

```
<jsxmodule name="revers-o-matic" timeout="0" debug>
```

This tag does three things:

- It specifies the name of the module. If you do not specify the name here then GoLive uses the name of the folder containing your Main.html tag.
- It turns on debugging. This enables you to use GoLive's built-in debugger to check problems with the code.
- It sets the error timeout to 0. If you do not set this attribute, in some circumstances if your code contains an error GoLive may get stuck in an endless loop. This attribute forces the loop to time out, thus preventing a freeze.

Creating the menu

Now define a menu item. Because this Extend Script performs only one function, you only need one menu item and you have no need to create an entirely new menu. GoLive enables you to add items to the special menu, so that's what you'll do.

Add the following code to your document, below the `<jsxmodule>` tag:

```
<jsxmenubar>
  <jsxmenu name="special">
    <jsxitem name="reversomatic" title="Reverse text">
  </jsxmenu>
</jsxmenubar>
```

As you can see, the markup for creating a menu is fairly self-explanatory. The `<jsxmenubar>` tags surround the block of menu definitions. All menu definition tags in the Main.html file must be contained within the `<jsxmenubar>` tags. The `<jsxmenu>` tag defines an actual menu — usually this tag is used to create a standalone menu, but by specifying the name as "special" you are able to add the menu item to the existing Special menu. You cannot add menu items to any of the other standard menus.

The `<jsxitem>` tag defines the menu item itself. The name attribute will be used later by your JavaScript — it names the menu item object in GoLive's object model. The title attribute is the actual text that is displayed in the menu.

If you like, you can test the menu now — save the Main.html file, quit GoLive, and relaunch. If you go to the Special menu you should now see a menu called "Reverse text," as shown in Figure E-3 The menu will not do anything at this stage, but you're almost there!

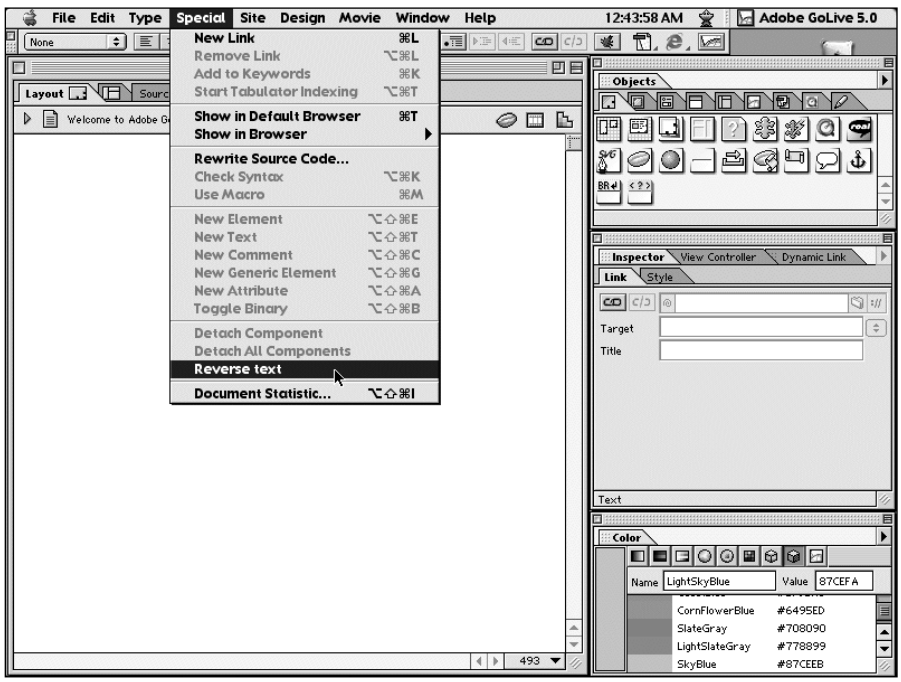

Figure E-3: The new menu item displayed in the Special menu

Handling the menu event

Now you need to tell GoLive what to do when the menu is selected. Reopen the Main.html file and switch to Layout mode. You will notice that the SDK markup tags are shown using the standard GoLive foreign tag display. Because you need to create some JavaScript code, drag the JavaScript item from the default tab of the Objects palette onto the page, ensuring that you place it between the `<jsxmodule>` and `<jsxmenubar>` blocks. Double-click the JavaScript icon to open the JavaScript editor window.

You don't need to use the JavaScript editor to edit your scripts—it's possible to edit them in Source view—but the script editor has some nice JavaScript editing features such as syntax coloring and error checking that make editing and debugging your code a lot easier.

When a menu item is selected, GoLive calls the `<menuSignal>` function. To define the function, enter the following code into the JavaScript Editor:

```
function menuSignal(menuItem)
{
  switch (menuItem.name)
  {
  case "reversomatic":  checkselection(); break;
  default: alert ("Sorry, an error occurred.");
  }
}
```

When GoLive calls the function, it also sends the argument `menuItem`. This is a standard GoLive menu item object, and therefore it has a `name` property, which you test for using the switch statement. The switch statement checks to see if `menuItem.name` is "reversomatic." If you recall, this is the name you gave the menu item in the `<jsxitem>` tag. If the name of the menu is indeed "reversomatic," then the function `checkselection()` is called; otherwise, an error message is displayed.

Checking that the selection is valid

The `checkselection()` function is required because the script is only valid if the user has made a text selection. You obviously don't want to try and run a text manipulation function on a table or an image, so you need to check that the current selection is valid text. The following code will do exactly that, so add it to your module:

```
function checkselection()
{
  //ensure that the user has selected text and not an image
etc.
  if (document.selection.element.elementType != "text")
  {
    alert("Please select a single run of text.");
```

```
    return false;
  }
  switch (document.selection.type)
  {
  // check the selection type

    case "full":  var el = document.selection.element;
                  var strt=document.selection.start;
                  var len=document.selection.length;
                  reverseText(el,strt,len);
                  break;
    case "none":  alert("You have not selected anything.");
                  break;
    case "point":  alert("You have not selected anything.");
                   break;
    case "complex":alert("Please select a single run of
text.");
                   break;
    case "part":  var el = document.selection.element;
                  var strt=document.selection.start;
                  var len=document.selection.length;
                  reverseText(el,strt,len);
                  break;
    case "outside":alert("You have just done the impossible!");
                   break;
    default:  alert("Sorry, an error occurred.");
  }
}
```

The first part of this function checks to see that the selected element is indeed text and not some other tag type. It does this by checking the value of the property `document.selection.element.elementType`. This is a standard property of the `element` object and can be any one of "text," "tag," "comment," or "bad." You are only interested in text selections, so you give the user an error message if the selection is anything else.

The next section of the function checks the type of selection the user has made. When you select an object in GoLive you can select all of the object, part of the object, or even multiple objects. You cannot run the text manipulation function if multiple objects are selected, nor can you run it if no objects are selected. You can test the `type` property of the `document.selection` object to determine what the current selection status is.

If the selection type is "full" or part — in other words, the user has selected either all the text or some of the text — then you call the function `reverseText()` and send it the selected element and the start and length of the current selection as arguments. If the user has made a "complex" selection (a multiple selection, including a multiple paragraph selection) or has not made a selection, then you present an error message using the standard JavaScript alert function.

Doing the work

Now that you know that you have a valid text selection, you can go about actually doing the text manipulation that the extension module is designed to do. All the actual work is done in the following function, which you should now add to your module code:

```
function reverseText(el,strt,len)
{

var newtxt="";

//break selection into component parts

  var txt=el.getInnerHTML();
  var reverse=txt.substring(strt,(strt+len));
  var lefttxt=txt.substring(0,strt);
  var righttxt=txt.substring((strt+len),txt.length);

  for (i=0; i<=reverse.length; i++)
  {
//loop through the text in reverse and store the result
 var newtxt = newtxt + reverse.charAt(reverse.length-i);
  }

  //replace the current text with our new text
  el.setInnerHTML(lefttxt+newtxt+righttxt);

  //force GoLive to reparse the document so our changes are
displayed
  document.reparse();
}
```

The function begins by creating a blank string variable to hold the new reversed string. Because the `setInnerHTML()` function that is used to write the new reversed text to the document replaces the whole selected element, we have to make sure that we only reverse the actual selected text. The function breaks the element's text into the text before the selection, after the selection, and the selection itself. It then loops through each character of the text selection in reverse and adds the characters to the new variable using the standard JavaScript text function `charAt()`.

The function then calls `setInnerHTML()`, which is a GoLive function that replaces the content of a markup element with a specified text string. In this case you are applying it to the element that contains the current text selection. This writes the new reversed string to the HTML document. Note that the element text is reconstructed with the reversed text in the middle.

Lastly, the function calls the `document.reparse()` function. This function causes GoLive to update the layout view based on the new markup.

That's it! You now need to close the JavaScript editor, save the page, quit GoLive, and relaunch it to try out your new extension.

Testing and debugging

If all goes well, you should be able to select some text on the page, go to the Special menu and select "Reverse text." If all goes really well, the text should then be reversed.

However, it's likely that a window popped up looking similar to Figure E-4. This is the JavaScript debugger, which you enabled in the `<jsxmodule>` tag. The bottom pane of the window displays your code, as shown in Figure E-4. The top-right pane shows debugging output, in this case the error that caused the debugger to display. The top-left pane is the stack trace that displays the calling hierarchy at the time the debugger was called. You can use the debugger to set breakpoints to make it easier to work out what part of your code is causing problems.

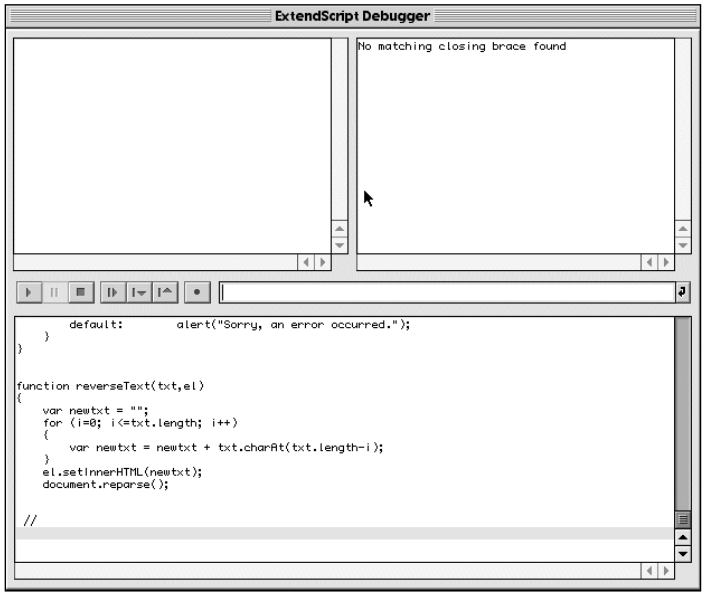

Figure E-4: The JavaScript Debugger window

If the debugger did display for you, it is likely that you have made a typing error when entering the example code. JavaScript syntax is particularly specific and is case-sensitive, so ensure you have entered the example code correctly. You should be able to use the debugger window to locate where the error has occurred in your code so that you can fix it.

Learning More About Extend Script and SDK

This is just a small sample of the power of Extend Script and the SDK. Much, much more could be explained here — your next step should be to read the Adobe documentation and study the prewritten samples that Adobe supplies so that you can discover some of the other possibilities that the SDK offers.