

# E CARE MANAGED SCRIPTS TECHNICAL GUIDE



**MOTOROLA**

Copyright © 2002-2008 Motorola, Inc. All Rights Reserved.  
Last Modified June, 2008

**Copyright notice**

Copyright © 2002-2008 Motorola, Inc. v. 062008

All rights reserved.

This manual and any associated artwork, software, product designs or design concepts are copyrighted with all rights reserved. Under the copyright laws this manual or designs may not be copied, in whole or part, without the written consent of Motorola. Under the law, copying includes translation to another language or format.

Motorola, Inc.

Marketplace Tower

6001 Shellmound Street, 4th Floor

Emeryville, CA 94608

USA

# CONTENTS

<b>eCare Managed Scripts Technical Guide</b> .....	<b>6</b>
Overview of eCare Managed Scripts .....	6
The Managed Script Execution Process and Environment .....	7
Package Format and Content.....	7
The Managed Script Execution Cycle .....	7
Executing the Script.....	8
Script Output.....	9
Package Requirements .....	9
Creating Your Script .....	10
Writing Your Script .....	10
Preparing Required Support Files .....	11
Testing Your Script .....	13
Debugging Your Script.....	13
Packaging Your Script.....	14
Creating The Package Manifest .....	14
Creating the Managed Script JAR File.....	19
Sample Managed Script Packages.....	22
<b>JavaScript Windows Object Model Reference</b> .....	<b>23</b>
eCare Scripting Library Objects .....	23
Using Scripting Objects.....	23
FileSystem() .....	24
System() .....	27
Com().....	28
Registry().....	30
String() .....	33
Macros( <i>ctx</i> ) .....	34
GlobalContext( <i>rootDir</i> ) .....	35
ZipFolder() .....	36
ZipFile( <i>gctx</i> , <i>zipName</i> , <i>folders</i> ).....	37

	CabinetFile( <i>gctx, cabFileName, infTemplateFilename</i> ).....	38
	StopWatch( <i>name</i> ).....	39
<b>Methods</b> .....		<b>40</b>
	addPathSlash .....	40
	banner .....	41
	buildPath.....	42
	buildUnixPath.....	43
	concatArray .....	44
	copyFile.....	45
	copyTree.....	46
	createBackup .....	47
	createFolder .....	48
	deleteFile.....	49
	deleteFiles .....	50
	deleteFolder .....	51
	deleteTree .....	52
	divider .....	53
	echo.....	54
	encloseXml.....	55
	enumFiles.....	56
	enumSubDirs .....	57
	exec.....	58
	execStrict.....	59
	fileExists.....	60
	findNextIndexOf.....	61
	fixPathSlashes.....	62
	fixRelPath.....	63
	folderExists .....	64
	formatElapsedTime .....	65
	formatTime.....	66
	genUuid.....	67
	getAbsolutePathName.....	68
	getCurrentDirectory.....	69
	getFile .....	70
	getFileName .....	71
	getFileTimeStamp .....	72
	getFolder.....	73

getParentFolderName.....	74
getTime .....	75
getVar .....	76
hasWildCards .....	77
makeCapabilitiesElement .....	78
makeDivider .....	80
makeFileWriteable .....	81
makeFilesWriteable .....	82
modifyAttr.....	83
moveFile.....	84
msgBox .....	85
openTextFile .....	86
raise.....	88
readTextFile .....	89
repeatChar.....	90
reportError .....	91
scriptDir.....	92
setVar .....	93
toHex.....	94
trimPathSlash.....	95
wildcardToRegExpr .....	96
writeFile.....	97
zeroPad.....	98

# eCARE MANAGED SCRIPTS TECHNICAL GUIDE

This document is intended for technical staff who may need to create or modify Managed Scripts. It describes how Managed Scripts are deployed, the environment in which they are can be executed, how to test your Managed Scripts, and how to package your scripts for installation on an eCare server.

These steps are discussed throughout this document.

## OVERVIEW OF eCARE MANAGED SCRIPTS

Because eCare Managed Scripts merely expose the remote computers native scripting capabilities, eCare's Managed Script capability is primarily a deployment and launch mechanism. This mechanism uses a dynamically deployed applet—the ScriptRunner applet—to retrieve (from the server), validate, and unpack special archive packages—the Managed Scripts. Once a package has been retrieved, validated, and unpacked on the remote computer, the ScriptRunner applet then invokes a fixed entry point to launch the script.

While this entry point varies by platform, it is always an OS-level job-scripting (batch) language running in a process separate from the Web browser process. This allows the package to invoke whatever specific executable or alternate scripting environment is best suited to the task at hand. In all cases, however, the script must behave as a simple console application. (The deployer applet will capture standard out from the script's process and return it to the server.)

In Windows, for example, the main entry point is a batch (CMD) file. In most of the Motorola-provided sample scripts, this batch file invokes `cscript` (the Windows Scripting Host) to execute either a JavaScript or VBscript program.

A Managed Script package is literally a JAR file with a special manifest entry. While JARs are most commonly associated with Java libraries and self-executing Java applications, typical eCare script packages do not include any Java—though Java applications can also be deployed and executed by this mechanism.

Managed Scripts use the JAR format because it has a well-defined standard for describing meta data, and it is supported with a readily available tool set.

## THE MANAGED SCRIPT EXECUTION PROCESS AND ENVIRONMENT

Before you create a Managed Script, it is important to understand the process by which Managed Scripts are executed on the customer's computer.

### PACKAGE FORMAT AND CONTENT

Your Managed Script must be packaged as a JAR file. While JARs are most commonly associated with Java libraries and self-executing Java applications, the JAR specification itself describes only a packaging standard.

For Managed Scripts that will run on Windows computers, the JAR file must contain the *scriptRunner.cmd* file, which is the predefined entry point described in the following section, [“The Managed Script Execution Cycle.”](#) It must also contain a manifest file that meets the requirements described in [“Creating The Package Manifest” on page 14.](#)

### THE MANAGED SCRIPT EXECUTION CYCLE

All of eCare's Managed Scripts are deployed and launched by eCare's ScriptRunner deployer applet.

When a Support Agent executes a Managed Script, the eCare server directs the ScriptRunner applet to retrieve the requested Managed Script package. The ScriptRunner applet downloads the Managed Script package to the customer's computer, expands the script package into a local directory, and then executes a predefined common entry point.

### THE COMMON ENTRY POINT: THE SCRIPTRUNNER.CMD FILE

In Windows, the primary entry point is the batch file *scriptRunner.cmd*. The deployer applet will execute this batch file in a separate process and capture that processes standard out and standard error channels.

The file name *scriptRunner.cmd* is hard-wired into the Windows ScriptRunner applet and cannot be changed. You may change other file names in the package as desired, provided you make the appropriate changes in the invoking file.

The entry-point batch file can be the entire “script,” or it can contain a command that will begin the execution of an alternate scripting engine. For example, many of the Windows example packages invoke `wsh` (using `cscript`) on the Windows Script Host (WSH) job file *scriptRunner.wsf*.

## USING THE SCRIPTRUNNER.WSF FILE

In the Windows environment, the WSF file is particularly important to other scripting languages, as it “sets up” the execution environment for the main script. Therefore, if you want to maintain support libraries and other code modules in separate files, you must provide a WSF file.

Note also that if you follow the standard launch practice of using *scriptRunner.cmd* to call a WSF file, WSH will execute the script in local mode. (All eCare Managed Scripts must be “local” scripts.)

As previously mentioned, the WSF file can also specify additional code. For example, Motorola provides the eCare JavaScript support library *Util.js*, which pre-defines access objects for key Windows resources such as the Registry, file system, and certain system-level resources. When *Util.js* is referenced in the WSF file, its objects will become part of the script’s object model at runtime.

## EXECUTING THE SCRIPT

When the ScriptRunner applet executes the Managed Script, it expects it to behave as simple console applications, without arguments. Results should be written to `standard out`.

Managed Scripts run with the Windows privileges assigned to the local user of the target computer. If the local user does not have Administrator-level privileges, the script will not be able to perform administrator- or system-level operations.

Under Windows Vista, eCare version 5.2 Managed Scripts are limited by both the client’s permission level and the browser’s integrity level—usually Medium integrity.

## SCRIPT OUTPUT

As the Managed Script runs, the ScriptRunner applet collects any output (both `standard out` and `standard error`, without interpretation) and returns it to the eCare server. The script result is presented to the eCare Support Agent in the results window.

The customer or Support Agent may also view the results later by clicking on the link in the eCare session transcript. The transcript file is saved in XML format, and two CSS style sheets are linked to the file. You may apply your own styles to the display of the transcript file by saving a CSS file called *custom-script-results.css* in the branding path on your eCare server. (Contact your eCare hosting administrator if you need to upload this file.)

In addition to this presentation, the output is automatically wrapped in minimal identifying meta data (in XML format) and archived.

## PACKAGE REQUIREMENTS

Minimally, a Managed Script package must meet three requirements:

- It must be a properly formatted JAR file.
- It must contain the *scriptRunner.cmd* file to function as the package entry point for a script that will run on Windows.
- The manifest file must contain a “package” entry that conforms to the requirements in [“Creating The Package Manifest” on page 14](#).

Thus, technically, the JAR must contain only two files: the manifest and the entry-point batch file. In certain cases, these two files will be all that your Managed Script requires.

However, the JAR file may contain any other files you wish. For example, you can include full executables, installer packages, support libraries, data files, and so on. This capability allows the Managed Script to act as a general purpose transport and deployment mechanism—your Managed Script package can include whatever you want to deploy and whatever it takes to properly deploy it. eCare will move the full package to the remote host, on demand, and then invoke your script to “install” it.

## CREATING YOUR SCRIPT

A discussion of writing scripts and the nearly infinite possibilities they can provide is beyond the scope of this document. You can download sample Managed Scripts in the eCare Administrator Center; the sample scripts are located on the *Managed Scripts* page in the eCare Configuration Manager. You may wish to examine these scripts for ideas and examples of proper script format.

To examine the components of a script package, run the following command in the folder that contains the JAR file.

```
jar xf SampleScript.Jar
```

The contents of the JAR file will be extracted in the same folder.

You may also rename the JAR file with the ZIP file extension. You can then open the ZIP file with any ZIP application and extract its components.

The following sections discuss the process of writing a WSH script (VBS or JavaScript) for deployment on a Windows host. Most of the sample scripts available in the eCare Configuration Manager are written and structured as discussed in these sections.

## WRITING YOUR SCRIPT

The main component of your eCare Managed Script is, of course, the script itself. For Windows computers, the script file will usually be written in JavaScript or VBS. However, a separate script file is not required for successful usage of a Managed Script. Small scripts may be embedded in the required WSF file (discussed in the following section, [“Preparing Required Support Files”](#)).

When you write your script, keep the following points in mind.

- Your WSH script will execute in local mode on the remote computer. Therefore, you should write all eCare Managed Scripts as “local” scripts. It is not necessary (and in most cases, not possible) to write a Managed Script as a WSH “remote” script.
- The ScriptRunner applet expects all Managed Scripts to behave as simple command-line tasks. Your scripts should run without arguments and write any important results to `standard out`.

- In eCare version 5.2, Managed Scripts run with the Windows privileges assigned to the local user of the target computer. If the local user does not have Administrator-level privileges, the script will not be able to perform administrator- or system-level operations.
- JavaScript scripts also have access to a library that creates a more convenient object model for many common diagnostic tasks. See the [“JavaScript Windows Object Model Reference” on page 23](#) for more information about the JavaScript library.
- To allow the greatest control over presentation, and to support later analysis, Motorola recommends that your scripts write their output as XML whenever possible. However, the script output should not be a full XML document; it should consist only of one or more well defined top-level XML elements. The eCare server will “wrap” the output with a document header and other meta data.

## PREPARING REQUIRED SUPPORT FILES

In addition to the script itself, your Managed Script package will require certain support files. Depending on your script, however, you may not need to modify the files—only include them when you create the script package. These files are discussed throughout this section.

### THE SCRIPTRUNNER.CMD FILE

In addition to the script file itself, your Windows script package must contain the *scriptRunner.cmd* file.

The easiest way to create this file is to extract it from an existing Managed Script package and modify it as needed for the Managed Script you are creating. (To extract the files, run the `jar xf` command or rename the file as described at the beginning of the section [“Creating Your Script” on page 10](#).)

The *scriptRunner.cmd* file must be located in the root directory of your Managed Script package. This file is the “known entry point” which the ScriptRunner applet will launch after the script package has been validated. This file **must** be called *scriptRunner.cmd*. If your script will execute in WSH and you supply a *scriptRunner.wsf* job file, you will not need to modify the sample *scriptRunner.cmd* file.

## THE SCRIPTRUNNER.WSF FILE

If you wish to execute your script with the Windows Scripting Host, you will most likely use a WSF file. In most of the sample eCare Managed Scripts, this file is called *scriptRunner.wsf*.

For simplicity, the *scriptRunner.wsf* file is located in the root directory of the sample Managed Script packages. The *scriptRunner.cmd* file calls this file, which then launches the Managed Script. The actual command (or set of commands) to launch the script should reside in the WSF file. In most eCare Managed Scripts, the WSF file calls the `Main` method in the JavaScript or VB-format script file.

If you wish to break your script into modules—for example, multiple JS files or a main program that uses one or more libraries—the WSF file is required. Or you may forgo a separate script file and include a small script within the WSF file itself. Consult the Microsoft WSH documentation for more information about the Windows Script Host and its capabilities.

**Note:** If you retain the *scriptRunner.wsf* file name, you will not need to modify the *scriptRunner.cmd* file. However, the Managed Script system does not depend upon or require that name for the WSF file.

## THE MANIFEST FILE

The JAR package that contains your Managed Script must include a manifest file, which must reside in the *META-INF* directory. The manifest file must meet the requirements discussed in detail in [“Creating The Package Manifest” on page 14](#).

## ADDITIONAL SUPPORT FILES

The JAR file may also contain any other files you wish to include, such as library files or other resources that your specific script requires.

Several of the sample Managed Scripts use the eCare *Util.js* JavaScript library. This library extends the WSH object model by defining additional objects, such as registry and file-system objects, that you can use to manipulate Windows resources. For more information about the objects and methods available with the *Util.js* file, see [“JavaScript Windows Object Model Reference” on page 23](#).

## TESTING YOUR SCRIPT

Before you package your script, make sure that it will run correctly in a simulated Managed Script environment.

Gather your script and its associated files, as listed in [“Preparing Required Support Files” on page 11](#). (Note that the manifest file is not required for testing.)

Place the contents of the script package in a local directory. This directory (or a subdirectory) should include all the files your script will expect when it is deployed. Be sure to include any libraries, such as the standard eCare JavaScript library, and any other included files, such as the replacement file that defines test values for the Ticket object.

On the command line, CD to the directory containing your script files. Then (for Windows scripts) run the command

```
cmd scriptRunner.cmd
```

If the *scriptRunner.cmd* file in your Managed Script only calls the *scriptrunner.wsf* file, you may use the following command instead.

```
cscript scriptRunner.wsf
```

This command duplicates the action of the *scriptrunner.cmd* file after it is invoked by the ScriptRunner applet.

Your script should run normally and return the expected output.

## DEBUGGING YOUR SCRIPT

If your script does not run correctly, you may single-step and inspect the execution of the script with Visual Studio or the Microsoft script debugger. Enable script debugging with the *Tools* ▶ *Options* menu in Internet Explorer, and then launch the script in debug mode. The simplest way to do so is to use the `//X` option:

```
cscript //X scriptRunner.wsf
```

## PACKAGING YOUR SCRIPT

Once you have created and tested your script, you are ready to create the manifest file and package your Managed Script as a JAR file.

Remember that your script package must include both the script itself and its support files, as listed in [“Preparing Required Support Files” on page 11](#). Be sure to include any libraries, such as the standard eCare JavaScript library, and any other included files.

### CREATING THE PACKAGE MANIFEST

Your Managed Script package must include a *manifest* file. When you create a JAR file with the JAR tool, the tool will automatically generate a default manifest with a single main section. However, the manifest for your Managed Script must include more than the default section.

Unfortunately, the standard JAR tool set does not allow you to edit and reuse the default manifest directly. To add the Managed Script information to the JAR’s manifest, you must manually create a text file with the additional manifest information.

You will then use a two step process to create the finished JAR file. The first step—a `jar cf` command—creates the JAR with a default manifest. The second step—a `jar umf` command—adds the Managed Script information to the JAR package. This process is described in [“Creating the Managed Script JAR File” on page 19](#).

### STANDARD MANIFEST REQUIREMENTS

A standard manifest file consists of an arbitrary number of name-value pairs called “attributes,” grouped into related sections. The sections in the manifest file have the following format requirements.

- All sections other than the main section must begin with a `Name` attribute.
- Each section must be separated from other sections by a blank line.
- Each section must contain one or more attribute-value pairs.
- Attribute values must not be empty. If a required attribute has no value, enter `NULL`.
- All attribute names must be terminated with a colon (:).

- All lines must be less than 72 bytes long (in its UTF-encoded form). Note that the limit is in bytes, not characters. Keep this consideration in mind if you must use double-byte characters.
- Lines longer than 72 characters may be continued on additional lines. Start continuation lines with a single space. The space will be removed when the manifest file is parsed.
- The manifest file must end in a blank line.

The following example manifest file contains a default main section and two additional sections.

```
Manifest-Version: 1.0
Created-By: 1.3 (Sun Microsystems, Inc)

Name: common/class1.class
Attribute1: some string value

Name: common/class2.class
Attribute2: A different value
Attribute3: Yet another value
```

For more information about manifest files and their requirements, see the JAR File Specification at

<http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html>

## ADDING THE MANAGEDSCRIPT MANIFEST SECTION

In addition to the default main section, the Managed Script package's manifest must include a single additional section named `ManagedScript`. To add the `ManagedScript` section to the JAR's manifest, you will first manually create a text file with the additional manifest information.

The simplest way to create your secondary manifest file is to extract and edit a copy of the *manifest.mf* file from one of the sample Managed Scripts available from the *Managed Scripts* page in the eCare Configuration Manager.

To extract the *manifest.mf* file, run the `jar xf` command or unzip the JAR file as described at the beginning of the section “[Creating Your Script](#)” on page 10. The *manifest.mf* file is located in the JAR package's *META-INF* directory.

### TO CREATE THE MANIFEST.TXT FILE

1. Obtain a copy of the *manifest.mf* file.
2. Open the file in a text editor, such as Notepad or vi.

You may also begin with a blank file and enter all required arguments manually.

3. Delete the main section, which begins with the attribute called `Manifest-Version`.

In most cases, the main section includes only the `Manifest-Version` and `Created-By` attributes and their values.

4. Leave a blank line at the top of the file.
5. Edit the remaining lines so that the script-specific attributes have values appropriate for your script. See the following section, “[Required Managed-Script Attributes](#),” for information about the meaning of each attribute.

Attribute values must not be empty. If an attribute has no value, use `NULL` as the value.

6. Make sure there is a blank line at the end of the file.
7. Save the file as *manifest.txt*.

Your finished *manifest.txt* file will appear similar to the following example.

```
Name: ManagedScript
Specification-Title: eCare Managed Script
Specification-Version: 1.0
Specification-Vendor: Motorola, Inc.
Display-name: Dump Environment
File-name: DumpEnv.jar
Short-description: List Environment Variables (Current process)
Full-Description: List all environment variables and their values (current process.)
Compatible-platforms: MSIE+SP2+Win, NS6+Win
Required-Capabilities: NULL
Required-Permissions: NULL
Guid: Placeholder
```

Notice the use of a continuation line in the `Full-Description` attribute value.

**Note:** Your new *manifest.txt* file should include all of the attributes listed in the following section, “[Required ManagedScript Attributes](#).” (Attributes will not necessarily appear in the same order.) If any of the required attributes are missing, add them and specify an appropriate value before you proceed with packaging your script.

## REQUIRED MANAGEDSCRIPT ATTRIBUTES

The `ManagedScript` section in the manifest file must include the following required attributes. Each attribute and its value must be written in the format

```
attribute: value
```

Attribute values in the `ManagedScript` section must not be empty. If an attribute has no value—for example, if the Managed Script package will run on any Windows computer, and thus has no value for `Required-Capabilities`—use `NULL` as the value.

In addition, keep in mind the formatting requirements discussed in the previous section, “[Standard Manifest Requirements.](#)”

<b>Name</b>	A required value; it must be set to <code>ManagedScript</code> .
<b>Display-name</b>	A short name for the Managed Script, which will appear in the Remote Script Selection pop-up window in the eCare Support Agent interface and the authorization window on the customer’s computer.
<b>File-name</b>	The name under which the package will be saved on the eCare server. The <b>File-name</b> value cannot be the same as any other Managed Script package on your eCare server, including earlier versions of the same script. If you attempt to reuse an existing file name, the eCare server will not save the new script.
<b>Short-description</b>	A brief description for the Managed Script, which will appear in the Remote Script Selection pop-up window in the eCare Support Agent interface. The short description is limited to 80 characters.
<b>Full-Description</b>	A longer description for the Managed Script, which will appear to the customer in the authorization window. The full description does not have a technical limit.
<b>Compatible-platforms</b>	A comma-separated list of platform identifiers on which the Managed Script can be executed. Valid values for <b>Compatible-platforms</b> are the name attributes associate with <code>UserAgent</code> elements in the <i>Metae.xml</i> file.

**Required-Capabilities**

A comma-separated list of capabilities, such as support for ActiveX, that must be present on the customer's computer in order for the Managed Script to run.

**Required-Permissions**

A comma-separated list of permissions, such as `premium.agent`, that must be assigned to the Support Agent's eCare profile before the Support Agent is authorized to execute the Managed Script.

**Guid**

In new Managed Script packages, a placeholder; you may enter any non-empty value. The value will be replaced automatically when the Managed Script is uploaded to the eCare server.

In a Managed Script package that has already been installed on the eCare server—for example, one that you copied from another service—the **Guid** attribute will contain a unique identifier (computed by the eCare server) which eCare can use to recognize and validate the package. (This feature allows you to package the script once and use it with multiple eCare services.)

**Specification-Title****Specification-Version****Specification-Vendor**

Attributes that identify the standard authority for an eCare `ManagedScript` section. These attributes are included in the manifest to conform to the intent of the manifest specification. They should always be the same as in the sample manifest at the end of the procedure [“To create the manifest.txt file” on page 15](#).

**OPTIONAL MANAGEDSCRIPT ATTRIBUTE****Timeout**

An optional attribute that specifies the amount of time, in milliseconds, that the `ScriptRunner` applet waits for the Managed Script to complete. (The timeout period begins when the customer approves the script, not when the Support Agent executes it.) If the script does not return a

response before the timeout period elapses, the ScriptRunner applet will report a script failure.

If you do not specify a **Timeout** attribute in your manifest file, the default value is 45 seconds.

## CREATING THE MANAGED SCRIPT JAR FILE

Before you can upload your Managed Script to the eCare server, it must be packaged as a JAR file. Each Managed Script JAR file you upload will appear in the Remote Script Selection pop-up window as a single scripting action the Support Agent can execute on the customer's computer. Note, however, that a Managed Script package can execute a series of scripts or accomplish multiple tasks if it is built to do so.

While JARs are most commonly associated with Java libraries and self-executing Java applications, the JAR specification itself describes only a packaging standard.

**Warning:** While JAR files are in actuality ZIP files, not all ZIP utilities use a compression algorithm that is recognized by the standard Java JAR manipulation classes. For that reason we recommend that you use Sun's JAR tool to zip your script packages.

The following procedure is the simplest way to create a Managed Script JAR file in the correct format.

### TO CREATE THE JAR FILE

1. Create a temporary directory for the Managed Script package.
2. Within this directory, create a directory named *build*. Place all component files except the *manifest.txt* file in the *build* directory.

The *build* directory (or a subdirectory) should include all the files your script will expect when it is deployed. Remember that the JAR file you build must also contain all of the files listed in [“Preparing Required Support Files” on page 11](#).

3. If you have not already done so, create the *manifest.txt* file with a script-specific manifest section as described in [“Adding the ManagedScript Manifest Section” on page 15](#). Place it in the temporary directory as a sibling of the *build* directory.

For example, if you are creating a package named `HelloWorld`, your directory hierarchy should look like this.



4. On the command line, make certain the Java JAR tool is on your class path.
5. CD to the temporary directory you created (the directory that contains the *manifest.txt* file, not the *build* directory).
6. Issue the following two commands.

```
jar cf <jarfile name.jar> -C build .
jar umf manifest.txt <jarfile name.jar>
```

The first command creates a new JAR file. The JAR file contains a manifest file, which the JAR tool generates automatically. The second command adds the eCare-specific section from the *manifest.txt* file to the generated manifest.

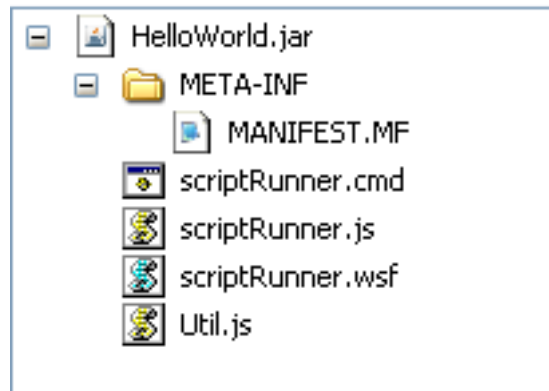
Note that

- In both commands, the JAR file name should include the *.jar* extension.
- The `-C` argument in the first command causes the JAR tool to change its working directory to *build*. Note that you **must** use a **capital C** in this command.
- The period at the end of the first command should be separated by a space from the word *build*. In this position, the period orders the JAR tool to include all files in the current directory when it creates the JAR file.

For example, in step 6 of the previous procedure, the following commands create a file named *HelloWorld.jar* in the *HelloWorld* directory.

```
jar cf HelloWorld.jar -C build .
jar umf manifest.txt HelloWorld.jar
```

The new JAR file has the following structure. (To view the file structure, run the `jar xf` command or rename the file as described at the beginning of the section [“Creating Your Script” on page 10](#).



Provided that the *manifest.txt* file was prepared properly (see [“Adding the ManagedScript Manifest Section” on page 15](#)), the generated manifest will be similar to the following example.

```
Manifest-Version: 1.0
Created-By: 1.6.0_01 (Sun Microsystems Inc.)

Name: ManagedScript
Guid: DumpEnvWinSp2Guid
Display-name: Dump Environment
File-name: DumpEnv.jar
Full-Description: List all environment variables and their values (current process.)
Required-Permissions: NULL
Specification-Vendor: Motorola, Inc.
Short-description: List Environment Variables (Current process)
Specification-Title: eCare Managed Script
Specification-Version: 1.0
Required-Capabilities: NULL
Compatible-platforms: MSIE+SP2+Win, NS6+Win
```

Note that when the JAR tool processes the manifest, it may reorder attributes (except for the leading Name attribute). It may also break long lines automatically, but do not assume that it will.

The script package is now ready to upload to the eCare server.

## **SAMPLE MANAGED SCRIPT PACKAGES**

Motorola has made several fully-packaged Managed Scripts available for you to use as-is or as a starting point for you to develop your own scripts. These sample scripts are available to download on the *Managed Scripts* page in the eCare Configuration Manager.

# JAVASCRIPT WINDOWS OBJECT MODEL REFERENCE

Motorola provides the eCare JavaScript support library *Util.js*, which pre-defines access objects for key Windows resources such as the Registry, file system, and certain system-level resources. This library of a includes a number of custom methods you can call against the scripting objects.

In addition to the scripting objects, many other JavaScript methods are made available with *Util.js*. These utility methods may simplify the execution of commonly used complex tasks. For a detailed discussion of the available methods, see [“Methods” on page 40](#).

## ECARE SCRIPTING LIBRARY OBJECTS

This section describes the scripting objects that are defined in the *Util.js* file. Each scripting object discussed in this document includes a list of the methods that can be called against it.

### USING SCRIPTING OBJECTS

To use a scripting object in your Managed Script, first create an instance of the object. Then call the methods on it. For example, the following command calls the **isEntryPresent()** method on the **Registry** object.

```
var registry = new Registry();  
registry.isEntryPresent( "\\HKEY_CURRENT_USER\\...");
```

## **FileSystem()**

The **FileSystem** object provides methods that can read and modify the computer's file system.

### **exists(*path*, *type*)**

Determines whether a file or directory is present. Specify the file or directory's path, as well as the file type, if relevant.

### **require(*path*, *type*)**

Returns an error if the specified file or directory is not present.

### **forbid(*path*, *type*)**

Returns an error if the file or directory is present.

### **force(*path*, *type*)**

Creates the file or directory specified by *path*, if the file or directory does not exist. If the file or directory exists, **force()** takes no action.

The *type* parameter specifies whether to create a file or a directory. Valid values for *type* are `File` and `Directory`.

### **remove(*path*, *type*)**

Deletes the file or directory specified by *path*, if the file or directory exists. If the file or directory does not exist, **remove()** takes no action.

The *type* parameter specifies whether the object to delete is a file or a directory. Valid values for *type* are `File` and `Directory`.

### **getFileSystemType(*filePath*)**

Indicates the file system type (for example, NTFS) of the drive on which the specified file is stored.

### **requireFile(*filePath*)**

Returns an error if the specified file does not exist.

### **isFilePresent(*filePath*)**

Indicates whether the specified file is present.

**IsFile(*path*)**

Indicates whether the object at the specified path is a file, and not a directory.

**createTextFile(*filePath*, *overwriteIfPresent*)**

Creates a file. This method can overwrite an existing file.

**writeTextFile(*filePath*, *fileData*,  
*overwriteIfPresent*)**

Writes the string *fileData* to the file specified by *filePath*.

If the file specified by *filePath* exists and *overwriteIfPresent* is set to `true`, `writeTextFile()` replaces the existing file. If the file exists and *overwriteIfPresent* is set to `false`, `writeTextFile()` takes no action.

**readTextFile(*filePath*)**

Retrieves the contents of a file.

**moveFile(*sourceFilePath*, *destination*)**

Moves the specified file to the specified location.

**copyFile(*sourceFilePath*, *destination*, *overwrite*)**

Copies the specified file to the specified location.

**deleteFile(*filePath*, *force*)**

Deletes a file.

**requireDirectory(*directoryPath*)**

Returns an error if the specified directory does not exist.

**isDirectoryPresent(*directoryPath*)**

Indicates whether the specified directory is present.

**isDirectory(*path*)**

Indicates whether the object at the specified path is a directory, and not a file.

**createDirectory(*directoryPath*, *overwriteIfPresent*)**

Creates a file. This method can overwrite an existing directory.

**moveDirectory(*sourceDirectoryPath*, *destination*)**

Moves the specified directory to the specified location.

**copyDirectory(*sourceDirectoryPath*, *destination*,  
*overwrite*)**

Copies the specified directory to the specified location.

**deleteDirectory(*filePath*, *force*)**

Deletes a directory.

**enumFiles(*directoryPath*, *deepEnum*)**

Lists the files in a specified directory. Use an optional Boolean `true` to list files within any sub-directories.

**numFiles(*directoryPath*, *deepNum*)**

Returns the number of files in the specified directory.

**enumDirectories(*directoryPath*, *deepEnum*)**

Lists the subdirectories within a specified directory.

**numDirectories(*directoryPath*, *deepNum*)**

Returns the number of subdirectories in the specified directory.

**getFSO()**

Returns the ActiveX object that is used to perform **FileSystem** operations.

## **System()**

The **System** object provides methods that can run shell commands.

### **exec(*command*, *wait*)**

Executes a command using the Windows Scripting Host (WSH).

### **remove(*text*, *regKey*, *mode*)**

Removes the specified registry key from the computer's registry.

You may optionally specify the removal mode as `quiet`. If you specify `quiet`, **remove** searches for a quiet removal method. If one is found, **remove** uses that method to remove the key. If no quiet method is found, **remove** uses the normal removal method.

## Com ( )

The **Com** object provides methods to determine whether a component is present and registered in the Windows registry.

**exists(*classId*, *progId*, *versionNum*,  
*registeringDllPath*)**

Determines whether the specified component is present and registered in the Windows registry.

**exists()** takes 4 parameters: *classId*, *progId*, *versionNum*, and *registeringDLLPath*. It verifies the component is registered by checking for

- The presence of *classId* under the registry key  
HKEY\_LOCAL\_MACHINE\SOFTWARE\Classes\CLSID\
- The presence of *progId.versionNumber* under the registry key  
HKEY\_LOCAL\_MACHINE\SOFTWARE\Classes\CLSID\classid\progid\
- The presence of *registeredDLLPath* under the registry key  
HKEY\_LOCAL\_MACHINE\SOFTWARE\Classes\CLSID\classid\  
InProcServer32\

**require(*classId*, *progId*, *versionNum*,  
*registeringDllPath*)**

Determines whether the specified component is present in the registry. If it is not present, **require()** returns an error.

**forbid(*classId*, *progId*, *versionNum*,  
*registeringDllPath*)**

Determines whether the specified component is present in the registry. If it is present, **forbid()** returns an error.

**remove(*classId*, *progId*, *versionNum*,  
*registeringDllPath*)**

**isComponentRegistered(*classId*, *progId*, *versionNum*,  
*registeringDllPath*)**

Determines whether the specified component is present and registered in the Windows registry.

**isComponentRegistered( )** takes 4 parameters: *classId*, *progId*, *versionNum*, and *registeringDLLPath*. It verifies the component is registered by checking for

The presence of *classId* under the registry key

HKEY\_LOCAL\_MACHINE\SOFTWARE\Classes\CLSID\

The presence of *progId.versionNumber* under the registry key

HKEY\_LOCAL\_MACHINE\SOFTWARE\Classes\CLSID\*classId*\*progId*\

The presence of *registeredDLLPath* under the registry key

HKEY\_LOCAL\_MACHINE\SOFTWARE\Classes\CLSID\*classId*\

InProcServer32\

## Registry()

The **Registry** object provides methods to create, enumerate, modify, and delete entries in the Windows registry. Other methods can check for the presence of a registry key or registry key value.

### **exists(text, regKey, regValue)**

Determines whether the specified registry key is present, and optionally whether it has the value specified by *regValue*.

### **require(regKey, regValue)**

Determines whether the specified registry key is present, and optionally whether it has a value. If the key or key-value pair is not present, **require()** returns an error.

### **forbid(regKey, regValue)**

Determines whether the specified registry key is present, and optionally whether it has a value. If the key or key-value pair is present, **forbid()** returns an error.

### **force(regKey, regValue, type)**

### **remove(regKey)**

Deletes the specified registry key.

### **isEntryPresent(regKey)**

Determines whether the specified registry key is present.

### **entryHasValue(regKey, regValue)**

Determines whether the specified registry key is present and has the value specified by *regValue*.

### **requireEntry(regKey)**

Determines whether the specified registry key is present. If it is not present, **requireEntry()** returns an error.

### **requireEntryHasValue(regKey, regValue)**

Determines whether the specified registry key-value pair is present. If it is not present, **requireEntryHasValue()** returns an error.

**getEntryValue(*regKey*)**

Retrieves the value of the specified registry key.

**getEntryValue()** calls the `RegRead` method on the WSH shell. Refer to the WSH documentation for more information about using the `RegRead` method.

**enumKeys(*regRoot*, *regBeginning*)**

Enumerates registry keys and returns them in an array.

Specify the registry tree that contains the path (the default is `HKEY_LOCAL_MACHINE`) and the path that contains the registry keys you wish to enumerate.

For example, the following code enumerates all the keys for the path `HKCU\MyNewKey\`.

```
registry.enumKeys("HKCU", "MyNewKey\\")
```

In the example section below, the only key returned will be `MyKey`.

**setEntryValue(*regKey*, *regValue*, *regType*)**

Creates the specified registry key or key-value pair. You may also use **setEntryValue()** to change the value of an existing registry key.

Optionally, you may include a *type* value, which specifies the data type of the registry key. Valid *type* keywords are `int` and `binary`. If you do not specify a *type* keyword, **setEntryValue()** assumes a string value.

**setEntryValue()** calls the `RegWrite` method on the WSH shell. Refer to the WSH documentation for more information about using the `RegWrite` method.

**deleteEntry(*regKey*)**

Deletes the specified registry key.

**deleteEntry()** calls the `RegDelete` method on the WSH shell. Refer to the WSH documentation for more information about using the `RegDelete` method.

**EXAMPLES**

```
var registry = new Registry();
registry.setEntryValue("HKCU\\MyNewKey\\", 1, "int")
registry.setEntryValue("HKCU\\MyNewKey\\MyKey", "Hello
    world!")

registry.getEntryValue("HKCU\\MyNewKey\\MyKey")
registry.getEntryValue("HKCU\\MyNewKey\\")

registry.deleteEntry("HKCU\\MyNewKey\\MyKey")
registry.delete("HKCU\\MyNewKey\\")
```

## **String()**

The **String** object provides methods to manipulate strings.

### **left(*n*)**

Returns the first *n* characters of the string. If the string has fewer than *n* characters, **left()** returns the entire string.

### **right(*n*)**

Returns the last *n* characters of the string. If the string has fewer than *n* characters, **right()** returns the entire string.

## **Macros(*ctx*)**

The **Macros** object is used to store an array of Macros for a specific context.

### **add(*name*, *value*)**

Adds the specified name-value pair as a macro to the **Macros** object.

## **GlobalContext (rootDir)**

The **GlobalContext** object allows you to create a context with respect to a specific directory, and then invoke methods using that directory as the “root” directory. Specify the root directory when you create the instance of the **GlobalContext** object.

### **getAbsPathFromRootRel (relPath)**

Returns the absolute path to the **GlobalContext** “root” directory, when given a specified path that is relative to that directory.

### **getAbsPathFromTargetRel (relPath)**

Returns the absolute path to the target directory, when given a specified path that is relative to that directory. The target directory is set by **setTargetDir ( )**, described below.

This method uses the **buildPath** method discussed on [page 42](#) to determine the absolute path.

### **getTargetDir ( )**

Returns the target directory.

### **getVersion ( )**

Returns version information for the specified file, as stored by **readVersionInfo ( )**.

### **readVersionInfo (fileName, verNum1Token, verNum2Token, verNum3Token, bldNumToken)**

Stores version information for the specified file. Specify the file name containing the version information, and pass in the version-information tokens as parameters.

### **setTargetDir (targetDir)**

Sets the target directory.

This method uses the **buildPath** method discussed on [page 42](#) to determine the absolute path.

## **ZipFolder()**

The **ZipFolder** object allows you to create a temporary ZIP folder, which is used to store files that will be added to a ZIP file. Each ZIP folder represents a folder in the ZIP file.

### **copyFiles(*gctx*, *tempDirName*)**

Specifies the path to the ZIP folder and a list of arguments, which specify the files that are to be contained in the ZIP folder.

**ZipFile(*gctx*, *zipName*, *folders*)**

The **ZipFile** object is used to create a ZIP file.

To construct a **ZipFile** object, pass in the **GlobalContext** object, the name of the ZIP file to create, and the list of **ZipFolder** objects that represent the folders to include in the ZIP file.

**create(*verbose*)**

Creates the ZIP file.

Optionally, you may set the Boolean *verbose* argument to `true` to echo the file-creation process to the console.

**EXAMPLE**

```
var zip = new ZipFile (
    gContext,
    "$(TargetDir)subDir\\zipFile.zip",
    // each ZipFolder represents a folder in the zip
    // file. The list of file for each folder tells
    // where to pull the content.
    [
        new ZipFolder("zipRelPath\\forZipTree",
            "$(RootDir)subDir\\File1.ext1",
            "$(TargetDir)subDir\\File2.ext2",
            ...),
        new ZipFolder("zipRelPath\\forZipTree2",
            "$(TargetDir)subDir2\\File3.ext3",
            "$(RootDir)subDir2\\File4.ext4",
            ...),
        ...
    ]
);
zip.create();
```

**CabinetFile(*gctx*, *cabFileName*,  
*infTemplateFilename*)**

The **CabinetFile** object is used to create a cabinet (CAB) file.

To construct a **CabinetFile** object, pass in the **GlobalContext** object, the name of the CAB file to create, and the list of folders to include in the CAB file.

**create()**

Creates the CAB file.

## **StopWatch( *name* )**

The **StopWatch** object provides methods that emulate a stopwatch. To construct a **StopWatch** object, pass in a name for the stopwatch.

### **getElapsed( )**

Returns the number of milliseconds elapsed since the object was created.

### **getElapsedStr( *useHours* )**

Returns the number of milliseconds elapsed since the object was created as a string.

If the Boolean *useHours* argument is `true`, and one or more hours has elapsed, **getElapsedStr( )** returns the elapsed time in `H* :MM :SS` format, where `H*` can be any integer greater than 0. (`H*` is not restricted to the range 0–23.)

If the *useHours* argument is `false`, or less than one hour has elapsed, **getElapsedStr( )** returns the elapsed time in `M* :SS` format, where `M*` can be any integer greater than or equal to 0. (`M*` is not restricted to the range 0–59.)

### **report( *useHours* )**

Calls the **getElapsedStr( )** and prints the result to the console. **report( )** does not return a value.

# METHODS

The following utility functions are defined at the global level.

## **addPathSlash**

Adds an additional backslash (\) at the end of the path, if one is not already present.

### **USAGE**

```
function addPathSlash (sPath)
```

### **ARGUMENTS**

**sPath** Required. Specifies the path.

## banner

Echoes specified text, surrounded by a banner.

### USAGE

```
function banner (sText, theChar, length)
```

### ARGUMENTS

- |                |  |
|----------------|--|
| <b>sText</b>   | Required. The text to appear within the banner.  |
| <b>theChar</b> | Optional. The character used to create the banner. If this argument is not present, the asterisk character (*) is used by default. |
| <b>length</b>  | Optional. The number of characters to use in the banner. If this argument is not present, the default is 80.                       |

For example, the method

```
banner("TheTextToSurround" , "*" , 30)
```

will produce the following output.

```
*****
TheTextToSurround
*****
```

## buildPath

Appends a file name or folder name to the specified path. Calls the `buildPath()` method on the `FileSystem()` object.

### USAGE

```
function buildPath (sPath, sName)
```

### ARGUMENTS

<b>sPath</b>	Required. Specifies the path to which <i>sName</i> will be appended. The path can be absolute or relative, and it does not need to specify an existing directory.
<b>sName</b>	Required. The file name or folder name to append to <i>sPath</i> .

### REMARKS

This method inserts an additional path separator between the existing path and the name, if necessary.

## **buildUnixPath**

Appends a name to a specified UNIX path.

### **USAGE**

```
function buildUnixPath (cur, tail)
```

### **ARGUMENTS**

<b>cur</b>	Required. Specifies the path to which <i>tail</i> will be appended.
<b>tail</b>	Required. The name to append to <i>cur</i> .

### **REMARKS**

This method inserts an additional path separator between the current path and the tail, if necessary.

## **concatArray**

Concatenates arrays that are passed in as arguments into a single array.

### **USAGE**

```
function concatArray () // (array1, ...)
```

### **ARGUMENTS**

The argument list is variable. It may contain one or more array elements to be concatenated. For example, to concatenate arrays named **array1** and **array2**, use the following method.

```
concatArray(array1, array2)
```

The method will return a single array that combines the elements of **array1** and **array2**.

**Note:** Arguments need not be array objects. For example, if the second argument contains a string, the method will return a new array that contains the elements of the first array, plus the string value.

If all arguments are non-arrays, **concatArray()** creates a new array in which each argument is an entry.

## copyFile

Copies one or more files from one location to another.

### USAGE

```
function copyFile (sSrc, sDst, bOverwrite)
```

### ARGUMENTS

- sSrc** Required. Specifies the location and name of the file or files to copy. This argument may include wildcard characters in the specified file name.
- Wildcard characters may be used only in the file name component of the *sSrc* argument. For example, the following usage is acceptable.
- ```
copyFile ("c:\\mydocs\\letters\\*.doc",
  "c:\\tempfolder\\")
```
- However, the following usage is not supported.
- ```
copyFile ("c:\\mydocs\\*\\mydoc.doc",
  "c:\\tempfolder")
```
- sDst** Required. The location to which the file or files specified by *sSrc* are to be copied. This argument may *not* include wildcard characters.
- The **copyFile()** operation will fail if *sDst* specifies a read-only file or folder, regardless of the value of *bOverwrite*.
- bOverwrite** Optional. A Boolean value that specifies whether existing files with the same file name should be overwritten. If this value is *true*, files will be overwritten. If this value is *false*, files will not be overwritten. If this argument is not present, the default is *true*.

**Note:** If the **gAnkVerboseFileSysCalls** global variable is set to *true*, **copyFile()** writes the source and destination file names to the console. (Its default setting is *false*.) To set **gAnkVerboseFileSysCalls**, include the statement `gAnkVerboseFileSysCalls = true;` in your script.

## copyTree

Copies all files in the source directory, including any sub-directories, to the destination directory. Specified files may be excluded from the copy operation.

### USAGE

```
function copyTree (sSourceDir, sDestDir, bEcho,  
                  exclude)
```

### ARGUMENTS

<b>sSourceDir</b>	Required. The source directory from which the files are copied.
<b>sDestDir</b>	Required. The destination directory to which the files are copied. If the directory does not exist, it is created. If a file with the same name as a file to be copied is present in the destination directory, the existing file is overwritten.
<b>bEcho</b>	Optional. If <code>true</code> , <code>copyTree()</code> writes the source and destination file names to the console.
<b>exclude</b>	Optional. Specifies files to omit from the copy operation. Specify <code>exclude</code> as a JavaScript regular expression, which may include wildcards. Matching files in both the source directory any sub-directories are omitted.

## createBackup

Backs up one or more files.

### USAGE

```
function createBackup (sFileName, sBackupName, bMove)
```

### ARGUMENTS

<b>sFileName</b>	Required. Specifies the location and name of the file or files to copy. This argument may include wildcard characters in the specified file name.
<b>sBackupName</b>	Required. Character string destination for the back up file. If a file with the specified file name is present in the destination directory, the existing file is overwritten.
<b>bMove</b>	Optional. A Boolean value that specifies whether the backup operation is treated as a move operation. If this value is <code>true</code> , the original file specified by <code>sFileName</code> will be deleted. If this value is <code>false</code> , the file will not be deleted. If this argument is not present, the default is <code>false</code> .

**Note:** `createBackup` calls the `moveFile` method if `bMove` is set, and the `copyFile` method if `bMove` is not set. Therefore, it can be used to back up multiple files. See [copyFile on page 45](#) and [moveFile on page 84](#) for more information about these methods.

## **createFolder**

Creates a folder in the specified location.

### **USAGE**

```
function createFolder (sPath)
```

### **ARGUMENTS**

<b>sPath</b>	Required. Specifies the location of the folder to create. If the specified path to the new folder does not exist, <b>createFolder</b> creates the necessary folders to complete the path. If the specified folder already exists, <b>createFolder</b> returns an error.
--------------	---

## **deleteFile**

Deletes the specified file.

### **USAGE**

```
function deleteFile (file, notIfReadOnly)
```

### **ARGUMENTS**

- file** Required. Specifies the location and name of the file or files to delete. This argument may include wildcard characters in the specified file name.
- Wildcard characters may be used only in the file name component of the *file* argument. For example, the following usage is acceptable.
- ```
deleteFile ("c:\\mydocs\\letters\\*.doc")
```
- However, the following usage is not supported.
- ```
deleteFile ("c:\\mydocs\\*\\mydoc.doc")
```
- notIfReadOnly** Optional. A Boolean value that specifies whether a file should be deleted if has a read-only attribute. If this value is `true`, the file will not be deleted if the read-only attribute is present. If this value is `false`, the file will be deleted. If this argument is not present, the default is `true`.

If no matching files are found, **deleteFile()** returns an error and ends the operation. **deleteFile()** stops when it encounters its first error. It does not reverse any previous changes.

## **deleteFiles**

Deletes the files in the specified directory.

### **USAGE**

```
function deleteFiles (sDir)
```

### **ARGUMENTS**

<b>sDir</b>	Required. Specifies the directory that contains the files to be deleted. If a file in the directory has a read-only attribute, the file is not deleted.
-------------	--

## **deleteFolder**

Deletes the specified folder.

### **USAGE**

```
function deleteFolder (sFolderName)
```

### **ARGUMENTS**

**sFolderName** Required. Specifies the folder to delete.  
If the folder has a read-only attribute, the folder is not deleted.  
If the specified folder does not exist, **deleteFolder()** performs no action.

**Note:** **deleteFolder()** does not distinguish between folders that contain files or sub-folders and those that do not. The specified folder is deleted regardless of whether it contains any files or sub-folders.

## **deleteTree**

Deletes all files and folders in a specified directory, including any sub-directories and their files.

### **USAGE**

```
function deleteTree (sRoot, keepRoot)
```

### **ARGUMENTS**

<b>sRoot</b>	Required. The directory from which to delete the files. If the <i>sRoot</i> directory does not exist, this method returns.
<b>keepRoot</b>	Optional. A Boolean value that specifies whether the <i>sRoot</i> directory itself should be deleted. If this value is <i>true</i> , the directory will not be deleted. If this value is <i>false</i> , the directory will be deleted. If this argument is not present, the default is <i>false</i> .

## divider

Creates a divider string and prints it to the console.

### USAGE

```
function divider (theChar, length)
```

### ARGUMENTS

- |                |   |
|----------------|---|
| <b>theChar</b> | Optional. The character used to create the divider. If this argument is not present, the asterisk character (*) is used by default. |
| <b>length</b>  | Optional. The number of characters to use in the divider. If this argument is not present, the default is 80.                       |

## **echo**

Echoes specified text to the console.

### **USAGE**

```
function echo (sText)
```

### **ARGUMENTS**

**sText** Required. Specifies the text to echo.

## **encloseXml**

Creates and returns a string representing an XML element in the form `<tagName>content</tagName>`.

### **USAGE**

```
encloseXml(tagName, content, indent)
```

### **ARGUMENTS**

<b>tagName</b>	Required. Specifies the name of the XML tag.
<b>content</b>	Required. Specifies the content of the XML tag.
<b>indent</b>	Optional. Specifies a string to prepend to the XML element string.

For example, the following method returns the string `<foo>bar</foo>`.

```
encloseXml("foo", "bar")
```

The following method returns the string `...<foo>bar</foo>`.

```
encloseXml("foo", "bar", "...")
```

## **enumFiles**

Returns a JavaScript enumerator object containing the list of files in the specified directory.

### **USAGE**

```
function enumFiles (sDir)
```

### **ARGUMENTS**

**sDir** Required. Specifies the directory containing the files to listed.

## **enumSubDirs**

Returns a JavaScript enumerator object containing the list of sub-directories in the specified directory.

### **USAGE**

```
function enumSubDirs (sDir)
```

### **ARGUMENTS**

**sDir** Required. Specifies the directory containing the sub-directories to listed.

## **exec**

Executes the specified command in the specified working directory.

### **USAGE**

```
function exec (cmd, workingDir, verbose)
```

### **ARGUMENTS**

<b>cmd</b>	Required. Specifies the command to execute.
<b>workingDir</b>	Optional. Specifies the directory in which to execute the command. If no directory is specified, the current directory is used.
<b>verbose</b>	Optional. A Boolean value that specifies whether the command is displayed in the console. If this value is <code>true</code> , the command is displayed. If this value is <code>false</code> , the command is not displayed. If this argument is not present, the default is <code>false</code> .

To return a detailed message when the command fails, use **execStrict()**.

## **execStrict**

Executes the specified command in the specified working directory. If the command fails, **execStrict()** returns a detailed message about the failure.

### **USAGE**

```
function exec (cmd, workingDir, verbose)
```

### **ARGUMENTS**

<b>cmd</b>	Required. Specifies the command to execute.
<b>workingDir</b>	Optional. Specifies the directory in which to execute the command. If no directory is specified, the current directory is used.
<b>verbose</b>	Optional. A Boolean value that specifies whether the command is displayed in the console. If this value is <code>true</code> , the command is displayed. If this value is <code>false</code> , the command is not displayed. If this argument is not present, the default is <code>false</code> .

To return no message when the command fails, use **exec()**.

## **fileExists**

Returns `true` if a specified file exists; `false` if it does not.

### **USAGE**

```
function fileExists (sFileName)
```

### **ARGUMENTS**

**sFileName** Required. Specifies the file whose existence is to be determined.

By default, **fileExists()** searches the current directory. To search a different directory, specify an absolute or relative path in addition to the file name.

## **findNextIndexof**

Returns the index position of the specified substring within the specified string.

### **USAGE**

```
function findNextIndexof (s, c, n)
```

### **ARGUMENTS**

- s** Required. Specifies the string in which to search for the specified substring.
- c** Required. Specifies the substring for which to search.
- n** Required. Specifies the index position within the string from which the search begins.

If the specified substring does not occur, **findNextIndexof()** returns `-1`.

## **fixPathSlashes**

Replaces incorrect path-separator characters in the specified path with the correct path-separator character and returns the result. **fixPathSlashes()** recognizes forward slashes and backslashes as path-separator characters.

### **USAGE**

```
function fixPathSlashes (sPath, sGoodChar)
```

### **ARGUMENTS**

- sPath** Required. Specifies the path in which path-separator characters will be corrected.
- sGoodChar** Optional. Represents the correct character to use in the specified path. The default value is `\\`.
- If *sGoodChar* is set to the forward slash character (`/`), **fixPathSlashes()** replaces any backslash characters in *sPath* with slashes. Conversely, if *sGoodChar* is set to the backslash character (`\\`), **fixPathSlashes()** will replace any forward slash characters in *sPath* with backslashes. (Note that you must escape the backslash character if you specify it.)
- If *sGoodChar* is any character other than a forward slash or backslash, **fixPathSlashes()** replaces all slashes with *sGoodChar*. If *sGoodChar* is null, **fixPathSlashes()** replaces forward slashes with backslashes.

## **fixRelPath**

Returns the absolute path to the specified relative path.

### **USAGE**

```
function fixRelPath (s)
```

### **ARGUMENTS**

**s** Required. Specifies the path, relative to the path on which the script is being executed. Example

For example, if the script is executed on  
`c:\path\scripts`  
and the method is  
`fixRelPath("relative\test.txt")`  
the output will be  
`c:\path\scripts\relative\test.txt`

## **folderExists**

Returns `true` if a specified folder exists; `false` if it does not.

### **USAGE**

```
function folderExists (sFolderName)
```

### **ARGUMENTS**

**sFolderName** Required. Specifies the folder whose existence is to be determined.

By default, **folderExists()** searches the current directory. To search a different directory, specify an absolute or relative path in addition to the folder name.

## **formatElapsedTime**

Returns the elapsed time between the specified *startTime* and *stopTime* in *H\* :MM :SS* or *M\* :SS* format.

### **USAGE**

```
function formatElapsedTime (startTime, stopTime,  
                             useHours)
```

### **ARGUMENTS**

<b>startTime</b>	Required. Specifies the number of milliseconds elapsed between January 1, 1970 and the actual start time.
<b>stopTime</b>	Optional. Specifies the number of milliseconds between January 1, 1970 and the actual stop time. If not specified, the current time is used.
<b>useHours</b>	Optional. A Boolean value that specifies whether the number of hours is displayed. If this value is <code>true</code> , the number of hours is displayed (in <i>H* :MM :SS</i> format, where <i>H</i> is any integer greater than 0). If this value is <code>false</code> , or if <i>H</i> is 0, the number of hours is not displayed (time is displayed in <i>M* :SS</i> format instead). If this argument is not present, the default is <code>false</code> .

For example, to return the elapsed time between January 1, 2000 and the current date, call the method as follows.

```
var d = new Date();  
d.setFullYear(2000, 01 ,01);  
var startTime = d.getTime();  
var stopTime = new Date().getTime();  
formatElapsedTime(startTime, stopTime, true);
```

## **formatTime**

Returns the specified elapsed time in `H* :MM :SS` or `M* :SS` format.

### **USAGE**

```
formatTime(t, useHours)
```

### **ARGUMENTS**

<b>t</b>	Required. Specifies the elapsed time in milliseconds.
<b>useHours</b>	Optional. A Boolean value that specifies whether the number of hours is displayed. If this value is <code>true</code> , the number of hours is displayed (in <code>H* :MM :SS</code> format, where <code>H</code> is any integer greater than 0). If this value is <code>false</code> , or if <code>H</code> is 0, the number of hours is not displayed (time is displayed in <code>M* :SS</code> format instead). If this argument is not present, the default is <code>false</code> .

## **genUuid**

Returns a universally unique identifier (UUID).

### **USAGE**

```
function genUuid ()
```

### **ARGUMENTS**

None.

## **getAbsolutePathName**

Returns a complete and unambiguous path from a provided path specification. This method calls the **GetAbsolutePathName()** method on the **FileSystem()** object.

### **USAGE**

```
function getAbsolutePathName (relname)
```

### **ARGUMENTS**

**relname** Required. Specifies the path to return as a complete and unambiguous path.

Note that the path is “complete and unambiguous” if it provides a complete reference from the root of the specified drive. A complete path can end with a path separator character (\) only if it specifies the root folder of a mapped drive.

## **getCurrentDirectory**

Returns the absolute path of the current directory.

### **USAGE**

```
function getCurrentDirectory ()
```

### **ARGUMENTS**

None.

## **getFile**

Returns a **File** object corresponding to the file at a specified path.

### **USAGE**

```
function getFile (sFile)
```

### **ARGUMENTS**

**sFile** Required. Specifies the path (absolute or relative) to the file. Wildcard characters are not supported.  
If the file does not exist, **getFile()** returns an error.

## **getFileName**

Returns the last component of the specified path that is not part of the drive specification.

### **USAGE**

```
function getFileName (sFullName)
```

### **ARGUMENTS**

<b>sFullName</b>	Required. Specifies the path (absolute or relative) to the file or folder.
------------------	--

**getFileName ( )** returns a zero-length string if the specified path does not include a component other than the drive specification.

## **getFileTimeStamp**

Returns the time the file was last modified.

### **USAGE**

```
function getFileTimeStamp (file)
```

### **ARGUMENTS**

<b>file</b>	Required. Specifies the file name. The time that <b>getFileTimeStamp()</b> returns is the number of milliseconds elapsed between midnight of January 1, 1970, and the time the file was last modified.
-------------	---

## **getFolder**

Returns a **Folder** object corresponding to the folder at a specified path.

### **USAGE**

```
function getFolder (sDir)
```

### **ARGUMENTS**

**sFile** Required. Specifies the path (absolute or relative) to the folder. Wildcard characters are not supported.  
If the folder does not exist, **getFolder ( )** returns an error.

## **getParentFolderName**

Returns the name of the folder *n* levels above the last component in the specified path.

### **USAGE**

```
function getParentFolderName (sPath, count)
```

### **ARGUMENTS**

- sPath** Required. Specifies the path to the component whose parent folder name is to be returned.
- count** Optional. If specified, **getParentFolderName()** counts *n* levels up from the specified component and returns the name of the folder at that level. If not specified, the default *n* is 1.
- getParentFolderName()** returns a zero-length string if there is no folder at the *count* level for the component specified in the *path* argument.

For example, the following method would return `Docs`.

```
getParentFolderName("C:/Docs/folder/test.txt", 2),
```

The following method would return `folder`.

```
getParentFolderName("C:/Docs/folder/test.txt")
```

## **getTime**

Returns the number of milliseconds since midnight of January 1, 1970.

### **USAGE**

```
function getTime ()
```

### **ARGUMENTS**

None.

## **getVar**

Returns the specified environment variable.

### **USAGE**

```
function getVar (name, defaultValue)
```

### **ARGUMENTS**

<b>name</b>	Required. Specifies the name of the environment variable to return. If the specified variable is not found, <b>getVar()</b> returns <i>defaultValue</i> .
<b>defaultValue</b>	Optional. Specifies the value to return if the <i>name</i> environment variable is not found. If <i>name</i> is not found, and <i>defaultValue</i> is not specified, <b>getVar()</b> returns null.

## **hasWildCards**

Returns `true` if the specified path includes one or more wildcard characters (\*).

### **USAGE**

```
function hasWildcards (sPath)
```

### **ARGUMENTS**

**sPath** Specifies the string representation of the path.

## makeCapabilitiesElement

Creates a string representing an XML element `<detected-capabilities>`.

### USAGE

```
makeCapabilitiesElement(caps, indent)
```

### ARGUMENTS

- caps** Specifies a JavaScript object in which each property is the name of a capability. `makeCapabilitiesElement()` uses the property names as capabilities and ignores the property values.
- The generated XML element will contain nested `<capability>` elements, one for each capability specified by `caps`. `makeCapabilitiesElement()` inserts a line-feed character (`\n`) after the opening `<detected-capabilities>` element and after each `<capability>` child element.
- indent** Optional. Specifies a string (usually blank spaces) to be prepended to each line in the result string.

For example,

```
var caps = new Object();
caps.foo = 'x';
caps.bar = 'y';
makeCapabilitiesElement(caps, "");
```

Returns

```
<detected-capabilities>\n
  <capability>foo</capability>\n
  <capability>bar</capability>\n
</detected-capabilities>
```

In addition,

```
var caps = {  
    baz: 'busy'  
};  
var indent = "....";  
makeCapabilitiesElement(caps, indent);
```

Returns

```
....<detected-capabilities>\n.....<capability>baz</capability>\n....</detected-capabilities>
```

## **makeDivider**

Creates a divider string.

Note that unlike `divider()`, `makeDivider()` returns a value.

### **USAGE**

```
function makeDivider (theChar, length)
```

### **ARGUMENTS**

- |                |   |
|----------------|---|
| <b>theChar</b> | Optional. The character used to create the divider. If this argument is not present, the asterisk character (*) is used by default. |
| <b>length</b>  | Optional. The number of characters to use in the divider. If this argument is not present, the default is 80.                       |

## **makeFileWriteable**

Removes the read-only attribute of the specified single file.

### **USAGE**

```
function makeFileWriteable (file)
```

### **ARGUMENTS**

**file** Specifies the name of the file whose read-only attribute is to be removed.

## **makeFilesWriteable**

Removes the read-only attribute of all files in the specified directory.

### **USAGE**

```
function makeFilesWriteable (sDir)
```

### **ARGUMENTS**

**sDir** Specifies the name of the directory that contains the files.

## **modifyAttr**

Modifies the attributes of the specified file.

### **USAGE**

```
function modifyAttr (fileOrName, mask, newValue)
```

### **ARGUMENTS**

<b>fileOrName</b>	Required. Specifies the name of the file.
<b>mask</b>	Required. Specifies the mask bit for <i>newValue</i> .
<b>newValue</b>	Required. Specifies the new value for the file attributes as a sum of values. For a list of available file attributes and their values, see <a href="http://msdn2.microsoft.com/en-us/library/5tx15443(VS.85).aspx">http://msdn2.microsoft.com/en-us/library/5tx15443(VS.85).aspx</a> <b>modifyAttr ( )</b> determines the value to set as the attribute by performing an AND operation on the <i>newValue</i> and <i>mask</i> .

### **EXAMPLE**

To set the “hidden” property of a file, call the method as

```
modifyAttr("testFile", 15, 2).
```

The value to set is calculated as  $15 \text{ AND } 2 = 2$ . To set the value equal to *newValue*, set the mask bit to contain all ones.

The “hidden” attribute is added to any existing attributes already present on the file.

## moveFile

Moves one or more files from one location to another.

### USAGE

```
function moveFile (sSrc, sDst)
```

### ARGUMENTS

**sSrc** Required. Specifies the location and name of the file or files to move. This argument may include wildcard characters in the specified file name.

Wildcard characters may be used only in the file name component of the *sSrc* argument. For example, the following usage is acceptable.

```
moveFile ("c:\\mydocs\\letters\\*.doc",
  "c:\\tempfolder\\")
```

However, the following usage is not supported.

```
moveFile ("c:\\mydocs\\*\\mydoc.doc",
  "c:\\tempfolder")
```

**sDst** Required. The location to which the file or files specified by *sSrc* are to be moved. This argument may *not* include wildcard characters.

The **moveFile()** operation will fail if *sDst* specifies a read-only file or folder.

**moveFile()** can move files between volumes only if supported by the operating system.

Note that if *sSrc* contains wildcards or *sDst* ends with a path separator (\), it is assumed that *sDst* specifies an existing folder to which the matching files will be moved. Otherwise, *sDst* is assumed to be the name of a destination file to create. In either case, three possibilities occur when an individual file is moved:

- If *sDst* does not exist, the file is moved. This is the usual case.
- If *sDst* is an existing file, **moveFile()** returns an error.
- If *sDst* is a directory, **moveFile()** returns an error.

If no matching files are found, **moveFile()** returns an error and ends the operation. **moveFile()** stops when it encounters its first error. It does not reverse any previous changes.

## **msgBox**

Opens a dialog box on the local computer. The script then waits for the local user to respond to the dialog box, or, optionally, for a specified timeout period to elapse.

### **USAGE**

```
function msgBox (sTitle, sText, nType, nWait)
```

### **ARGUMENTS**

<b>sTitle</b>	Optional. Specifies the title of the dialog box.
<b>sText</b>	Required. Specifies the text to display in the dialog box.
<b>nType</b>	Optional. Specifies the sum of values that controls the number and type of buttons to display in the dialog box, the icon to use, and the default button. For more information on available values for <i>nType</i> , see <a href="http://msdn2.microsoft.com/en-us/library/x83z1d9f.aspx">http://msdn2.microsoft.com/en-us/library/x83z1d9f.aspx</a>
<b>nWait</b>	Optional. Specifies the number of seconds the script will wait before it automatically closes the dialog box. If you specify a value for <i>nWait</i> and that timeout elapses when the script is run, <b>msgBox ( )</b> returns a value of -1.

## openTextFile

Opens the specified file and returns a **TextStream** object that can be used to read from, write to, or append to the file.

### USAGE

```
function openTextFile (sName, nMode, bCreate, nFormat)
```

### ARGUMENTS

<b>sName</b>	Required. Specifies the file to open.
<b>nMode</b>	Optional. Specifies one of three constants: <code>ForReading</code> , <code>ForWriting</code> , or <code>ForAppending</code> . See the <i>nMode Settings</i> table below for more information.
<b>bCreate</b>	Optional. A Boolean value that specifies whether a new file will be created if the specified <i>sName</i> does not exist. If this value is <code>true</code> , a new file is created. If this value is <code>false</code> , it no file is created. If this argument is not present, the default is <code>false</code> .
<b>nFormat</b>	Optional. Specifies one of three Tristate values to indicate the format of the specified file. If this argument is not present, the file is opened as ASCII. See the <i>nFormat Settings</i> table below for more information.

The *nMode* argument can have any of the following values.

nMode Values		
Constant	Value	Description
<code>ForReading</code>	1	Open the file for reading only. You cannot write to the file.
<code>ForWriting</code>	2	Open the file for writing.
<code>ForAppending</code>	8	Open the file and write to the end of the file.

The *nFormat* argument can have any of the following values.

nFormat Values		
Constant	Value	Description
TristateUseDefault	-2	Open the file using the system default.
TristateTrue	-1	Open the file as Unicode.
TristateFalse	0	Open the file as ASCII.

For example, the following method will open a file for appending text.

```
openTextFile("c:\\testfile.txt", ForAppending, false);
```

## **raise**

Returns an error.

### **USAGE**

```
function raise (desc, err)
```

### **ARGUMENTS**

<b>desc</b>	Required. Specifies the description of the error.
<b>err</b>	Required. Specifies the error to return. If <i>err</i> is an error object, <b>raise()</b> returns an error that includes the error number and appends the <i>err</i> description to <i>desc</i> .

## **readTextFile**

Returns the contents of the specified text file as an array, with each entry containing one line of text from the file.

### **USAGE**

```
function readTextFile (sFileName)
```

### **ARGUMENTS**

**sFileName** Required. Specifies the file to read.  
If the file does not exist, **readTextFile()** returns an error.

## **repeatChar**

Repeats the specified character the specified number of times.

### **USAGE**

```
function repeatChar (theChar, times)
```

### **ARGUMENTS**

- |                |   |
|----------------|---|
| <b>theChar</b> | Required. Specifies the character to repeat.  |
| <b>times</b>   | Required. Specifies the number of times to repeat the character.<br>If <i>times</i> is 0, <b>repeatChar</b> ( ) returns an empty character. |

## **reportError**

Returns the details of the specified error to the console. If the error object contains a number, it is reported along with the description of the error.

### **USAGE**

```
function reportError (e)
```

### **ARGUMENTS**

**e** Required. Specifies the error object.

## **scriptDir**

Returns the parent folder of the current script.

### **USAGE**

```
function scriptDir ()
```

### **ARGUMENTS**

None.

## **setVar**

Sets a new environment variable and value at the “Process” level.

### **USAGE**

```
function setVar (sVar, sValue)
```

### **ARGUMENTS**

<b>sVar</b>	Required. Specifies the name of the environment variable to set.
<b>sValue</b>	Required. Specifies the value of the environment variable.

## **toHex**

Returns the hexadecimal value of the specified number.

### **USAGE**

```
function toHex (n, pad)
```

### **ARGUMENTS**

- |            |  |
|------------|--|
| <b>n</b>   | Required. Specifies the integer in base 10.  |
| <b>pad</b> | Optional. Specifies the minimum length of the variable that is returned. If the <b>toHex()</b> operation results in a value of fewer than <i>pad</i> characters, additional “zero” (0) characters are appended to the beginning of the result to make a length of <i>pad</i> . |

## **trimPathSlash**

Removes any backslash (\) character from the end of the specified path.

### **USAGE**

```
function trimPathSlash (sPath)
```

### **ARGUMENTS**

**sPath** Required. Specifies the path.

## wildcardToRegExpr

Converts a string containing wildcard characters into a regular expression with proper escaping.

### USAGE

```
function wildcardToRegExpr (str)
```

### ARGUMENTS

**str** Required. Specifies the string upon which the conversion is to be performed.

For example, the method

```
wildcardtoRegExpr("abc$def?")
```

returns

```
abc\$.def.
```

## **writeFile**

Writes the specified content to the specified file.

### **USAGE**

```
function writeTextFile (sFileName, lines)
```

### **ARGUMENTS**

<b>sFileName</b>	Required. Specifies the file to which the content will be written.  If the file specified by <i>sFileName</i> already exists, its contents are overwritten. If it does not exist, a new file is created.
<b>lines</b>	Required. Specifies the content to add to the file.  <i>lines</i> may be an array or plain text. If it is an array, <b>writeFile()</b> inserts a new line character after it writes each entry of the array in the file. If it is plain text, <b>writeFile()</b> adds one new line character at the end of the text.

## zeroPad

Returns the specified variable with zeros prepended as required to make the variable a specified length.

### USAGE

```
function zeroPad (n, len)
```

### ARGUMENTS

- |            |  |
|------------|--|
| <b>n</b>   | Required. Specifies the variable to which padding is to be appended.   |
| <b>len</b> | Optional. Specifies the minimum length of the variable that is to be returned.<br><br>If the length of the variable <i>n</i> is less than <i>len</i> , additional “zero” (o) characters are appended at the beginning of the variable until its length equals <i>len</i> . If the length of the variable <i>n</i> is greater than or equal to <i>len</i> , or if <i>len</i> is not specified, <i>n</i> is returned without modification. |