equilibrium®

MEDIARICH®

MediaRich for SharePoint 2003 Programmer's Guide

v 3.6.194.0

Table of Contents

Chapter 1
Programming MediaRich for SharePoint

MediaRich for SharePoint is a client-server system that allows users to perform image processing tasks using media stored in Microsoft SharePoint Portal Server. It includes an "out-of-the-box" interface where users perform these tasks using either the Export interface, which is available from MediaRich Document Libraries, or the MediaRich MediaCart site, which is available from the MediaRich Browser.

To fit the needs of your organization and users, you can customize the MediaRich Document Libraries and MediaCart site using standard SharePoint tools and MediaRich's proprietary scripting language, MediaScript. This guide provides information about extending these interfaces by adding new MediaScripts.

For more information about the MediaRich for SharePoint architecture, deployment, and administration, see the *MediaRich for SharePoint Installation and Administration Guide*. And for more information about the default implementation of the MediaRich Document Libraries and MediaCart site, see the *MediaRich for SharePoint User Guide*.

Chapter Summary

MediaRich for SharePoint 2003 Features

File Format Support MediaRich for SharePoint extends SharePoint's standard picture library to provide previews of CMYK images, Vector images, and layered Photoshop® files along with related metadata. It reads and writes many popular file formats including BMP, WBMP, GIF, JPG, PNG, PCT, TIFF, PDF, TGA, Adobe Illustrator®, and Photoshop PSD or EPS files.

Workflow and Collaboration Users can ensure that the most current, approved image is used by taking advantage of SharePoint's routing and approval process, version control, and check-in/check-out system.

Search and Browse Users can search directly from a Microsoft Office application or within SharePoint Portal Server and Windows SharePoint Services sites. This includes viewing a thumbnail preview or zooming in for detailed inspection, regardless of the original file format. Users can also browse a hierarchical view of all the available document and picture libraries from a single Web Part.

Metadata Support MediaRich for SharePoint ingests metadata automatically when images are uploaded into the system. Users can view image properties, as well as IPTC and EXIF industry standard information, and they can easily update these fields, store the information in SharePoint, and write it back to the original files for future use.

Image Transformations MediaRich for SharePoint provides comprehensive image editing for image creation, modification, and delivery. It can automatically transform images using corporate guidelines, as well as provide the flexibility to select the file format, size, and resolution before downloading.

Batch Processing Any modification that needs to be made to a single image can be applied to an unlimited number of images. Administration controls allow users to view current job items, their status, and cancellation. In addition, the results can be returned back to SharePoint, sent to a FTP site, and sent using email with a link to the originals or as a compressed ZIP file.

Microsoft Office Support Users can preview individual pages of Word® and Excel® documents or full PowerPoint® presentations so that they find the exact asset that they are looking for. This helps to reduce time and bandwidth with unnecessary downloads and enables users to access individual slides to add to a presentation.

The Application Programming Framework

MediaRich image processing is controlled by MediaScript, which provides a simple framework that makes writing new scripts easy. The framework provides several distinct benefits:

- A clear API for interacting with the rest of the system
- Ability to develop individual scripts that can be focused on the processing required
- The same scripts can be used for both Export and the MediaCart

Working with MediaScript

MediaScript is an ECMAScript compliant language like JavaScript or JScript. Similar to those other implementations of ECMAScript, MediaScript makes available additional host objects relevant to the task at hand. This document assumes some familiarity with MediaScript. For more information about working with MediaScript, see Chapter 2, "Using MediaScript," and Chapter 3, "MediaScript Objects and Methods."

The most important new object in MediaScript is the Media object, which implements the many image processing features provided by MediaRich. A typical script creates one or more Media objects by either loading image data from a file or using the Media drawing methods. The script then performs additional image processing operations based on arguments and parameters in the request. Finally, the script generates a response by saving a Media object to the desired output file format.

This scripting framework provides a few key services:

- Loads the image that will be passed to a `processImage` function
- Calls the `processImage` function
- Saves the image returned from the `processImage` function

Additional steps are performed depending on whether the context is Export or the MediaCart.

Because this framework is provided as MediaScript, it is readily modifiable. However this requires advanced understanding of MediaScript. Before modifying the framework you should have some real experience writing MediaScript, read and understand the information about using MediaScript in general and in particular the sections on Media object, the Request object, the response object, and the TextResponse object.

There are separate scripts used to provide the framework for Export and the MediaCart:

For Export

```
[MediaRich Root]\Shared\Originals\Scripts\SharePoint\exportCrop.ms
```

For the MediaCart

```
[MediaRich Root]\Shared\Originals\Scripts\SharePoint\MRBatch.ms
```

NOTE: *These scripts are critical for users to be able to use the Document Export and MediaRich MediaCart site with SharePoint Products and Technologies. It is highly suggested that you ensure that there are back-up copies of these files before you modify either of them.*

Using the processImage Function

The MediaScript API for SharePoint is comprised of a single function your script must define. This enables the script to function correctly both as an Export script and a MediaCart script. In the case of the MediaCart, the same script is used for each image in a MediaCart batch.

Syntax

```
Media processImage(Media image)
```

The `processImage` function is the single entry point for all scripts that interoperate with the MediaRich for SharePoint Connector.

Parameters

The `processImage` function takes a Media object as its single argument. This object is an image from a Windows SharePoint Services site on which operations are performed.

Return

Your `processImage` function must return a Media object.

Example

The following sample script, named "`save image as jpg.ms`", specifies that an arbitrary input image should be saved using the JPG file format:

```
function processImage(image) {
    var saveParams = resp.getSaveParameters();
    saveParams.type = "jpg";
 resp.setSaveParameters(saveParams);
 return image; }
```

The first line:

```
function processImage(image) {
```

starts the definition of the `processImage` function. It is expecting to be passed a Media object named "image".

This script requires a little more information from the system, so the next thing to do is get this information:

```
var saveParams = resp.getSaveParameters();
```

This makes use of another object provided by the system, the Response object named "`resp`" that is available to all MediaScripts, and gets its save parameters.

The next line sets a save parameter:

```
saveParams.type = "jpg";
```

Setting the save parameter "type" determines what type of image the framework is going to save. The save parameters correspond to the parameters that can be passed to the Media object's `save()` method. For some formats, such as GIF and JPEG, there are many more parameters that can be specified.

NOTE: *For more information about what parameters are available for saving images, see "save()" on page 124.*

Before returning control to the framework, you must complete a final step:

```
resp.setSaveParameters(saveParams);
```

This line sets the save parameters with the instance altered in the script:

```
return image;
```

This last line returns the Media object named "image" for the framework to use in the response, applying the response's save parameters.

Deploying a New MediaScript

To deploy a MediaScript, copy the file to the appropriate directory. In a default installation, the directories are:

For Export

```
[MediaRich Root]\Shared\Originals\Scripts\SharePoint\Export
```

For the MediaCart

```
[MediaRich Root]\Shared\Originals\Scripts\SharePoint\Batch
```

NOTE: *These directories are set during the MediaRich for SharePoint (Server) installation. If you are not aware of the details of your installation, you should consult your System Administrator.*

Metadata Support

MediaRich fully supports loading, saving and merging IPTC, EXIF, and XMP metadata for JPEG, TIFF, and Photoshop files. MediaRich also supports loading XMP metadata from the following file formats: Illustrator, EPS, GIF, PDF, and PNG. This metadata is available to the script as a metadata XML document. Detailed schemas are provided for the EXIF and IPTC documents constructed by MediaRich. The XMP metadata document conforms to the schema defined by Adobe.

The MediaRich for SharePoint product currently supports only IPTC and EXIF metadata for JPEG, TIFF, and Photoshop files.

For for more information about the Metadata Interface and common methods, see Appendix A, "MediaRich Metadata Support."

Chapter 2
Using MediaScript

All MediaRich image templates are written in MediaScript, an interpreted scripting language based on the ECMAScript Language Specification, 3rd edition (which, in turn, is based on Netscape's JavaScript and Microsoft's JScript). By building on top of a widely known scripting language, MediaScript can offer all of the flexibility of a full programming language while remaining easy to use.

MediaScript supports all of ECMAScript's syntax and objects while adding several new objects and language enhancements. This chapter only describes those features unique to MediaScript; for a detailed description of the basic language, see the ECMAScript Language Specification (available at `http://www.ecma.ch/ecma1/STAND/ECMA-262.htm`) or one of the many available JavaScript references.

Chapter Summary

The Media Object

The most important built-in object in MediaScript is the Media object, which implements the many image processing features provided by MediaRich. A typical script creates one or more Media objects by either loading image data from a file or using the Media drawing methods. The script then performs additional image processing operations based on arguments and parameters in the request. Finally, the script generates a response by saving a Media object to the desired output file format.

For example, a script to scale an image might look like the following:

```
function main(imageName, width, height)
{
    var img = new Media();
    img.load(name @ imageName);
    img.scale(xs @ width, ys @ height, alg @ "best");
    img.save(type @ "jpeg");
}
```

For a complete description of all of the methods provided by the Media object, see Chapter 3, "MediaScript Objects and Methods."

Preprocessor Directives

Preprocessor directives are lines included in the code of our programs that are not program statements but directives for the preprocessor. These lines are always preceded by a pound sign (#) and are processed before the rest of the script and direct the way the script commands are interpreted.

These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is considered to end. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

The following statements are collectively called preprocessor directives, since they are processed before the rest of the script and direct the way the MediaScript commands are interpreted.

Using the #include Directive

The #include directive is a preprocessing flag that is evaluated before the script is parsed or executed. It allows other scripts to be included as if they were part of the original script and takes a string representing the relative path of the script filename to include. The string must be enclosed in double quotes.

Syntax

```
#include <"relative path to the included file">;
```

Example

```
#include "process/library.ms";
```

If your MediaScript uses the XMLdocument object, you need to use the #include directive to specify the `xml.ms` that installs with MediaRich. Do this by adding the following line at the beginning of your script:

```
#include "sys:xml.ms";
```

Using the #link Directive

The #link directive is a preprocessing flag that is evaluated before the script is parsed or executed. It allows DLLs to be included as if they were part of the original script and takes a string representing the relative path of the filename to include.

Syntax

```
#link <relative path to the included file>;
```

Example

```
#link <process/library.dll>;
```

If your MediaScript uses the Database, Cursor, or Stproc object, it must include the following line at the beginning of the script:

```
#link <Database.dll>;
```

Named Arguments

When reading the object reference section of this chapter, you will see that many of the Media object's methods use an argument notation of the form:

```
argName @ argValue
```

The reason for this notation is that these methods often have a long list of optional (and sometimes mutually exclusive) arguments. The `save()` method, for example, has sixteen arguments, many of which only apply to certain file formats. Moreover, adding a new file format plug-in to the system can add even more arguments. Because of this, certain methods take a set of name/value pairs rather than the standard positional argument list.

MediaScript introduces the at operator (`@`) to simplify named arguments, as in the following:

```
(argName1 @ argValue1, argName2 @ argValue2, …)
```

For example, if you have a Media object named “img” and you want to save the contents to a JPEG file named “out.jpg” at 90% quality, the MediaScript looks like the following:

```
img.save(name @ "out.jpg", quality @ 90);
```

All of the functions that use named arguments also accept a single object as an argument. If an object is passed, each of the object's properties corresponds to an argument name/value pair; the property's name is the argument name and the property's value is the argument value. For example, the previous MediaScript could be rewritten as the following:

```
var argsObject;
argsObject.name = "out.jpg";
argsObject.quality = 90;
img.save(argsObject);
```

The advantage of passing an object is that it can be reused across multiple calls.

As a final option, you can use ECMAScript's standard object literal syntax to pass a temporary, anonymous argument object. In this case, the MediaScript would be rewritten as:

```
img.save({ name: "out.jpg", quality: 90} );
```

File Systems

MediaRich for SharePoint server implements its own virtual file system for reading and writing data. This file system is fully customizable allowing advanced installations the ability of defining various input and output repositories for specific purposes.

MediaRich was designed to support large clusters of servers to maximize performance and reliability. There are two deployment scenarios for a MediaRich cluster. In one scenario, all of the servers share a common file system with shared properties, originals, and generated output files. In the other scenario, each MediaRich server maintains separate storage and an external mechanism is used to duplicate and synchronize originals across all of the servers. In either case, as a script developer you can write your scripts once without worrying about the details of each server's file system layout.

File System Specifiers

In order to abstract file system access, all paths in MediaScript can be prefixed by a file system specifier of the form:

```
specifier:/path
```

Specifiers must consist entirely of alphabetical characters; numbers and punctuation marks are not allowed. Most file system specifiers are simply aliases for local or shared directories. The following table lists these specifiers:

File System Specifier	Default Location
cache	`\MediaRich\Shared\Generated\MediaCache`
fonts	`\MediaRich\Shared\Originals\Fonts`
logs	`\MediaRich\Shared\Logs`
output	`\MediaRich\Shared\Generated`

File System Specifier	Default Location
profiles	\MediaRich\Shared\Originals\Profiles
read	\MediaRich\Shared\Originals\Media
results	\MediaRich\Shared\Generated\MediaResults
scripts	\MediaRich\Shared\Originals\Scripts
sys	\MediaRich\Shared\Originals\Sys
write	\MediaRich\Shared\Originals\Media

All MediaScript operations that take a path default to a reasonable file system when no specifier is present in the path. The following table lists these defaults:

MediaScript Operation	Default File System Specifier
Loading a script	scripts
Loading a color management profile	profiles
Saving a file to the MediaRich cache	results
All other read operations	read
All other write operations	write

For example, the following MediaScript:

```
img.load(name @ "bike.tif");
```

is equivalent to:

```
img.load(name @ "read:/bike.tif");
```

Some MediaScript file systems are not based on the standard disk file system. The `mem` specifier uses an abstract file system that is contained entirely in memory and is never written to disk, which is useful for fast access to temporary file data. The FSNet plug-in implements the `ftp` and `http` specifiers, which allow files to be accessed using FTP and HTTP URLs respectively.

Creating Custom File System Aliases

To define your own file system alias, all you need to do is create an entry in the `local.properties` file. Virtual file systems paths must start with an alphabetic character, but can then contain any alpha-numeric characters. For example, if you want MediaRich to read source media from a storage server with a UNC path of

"\\STOR1\source" and write the output to a web server with a UNC path of "\\WEB1\retail1\images", append the entries to your `local.properties` file, as in the following example:

```
virtualfilesystem.retailIN=\\\\STOR1\\source
virtualfilesystem.retailOUT=\\\\WEB1\\retail1\\images
```

NOTE *The backslash character (\) is the escape character for MediaRich properties files. All backslashes in the path must be escaped with an additional backslash.*

Before you can use the new file system aliases you must restart the MediaGenerator service.

The corresponding `load()` and `save()` operations would look like:

```
image.load(name @ "retailIN:/camera.psd");
image.save(name @ "retailOUT:/camera.jpg");
```

File Filesystem

The file filesystem allows standard "file" URLs to be supplied to MediaScript functions that take filenames. This filesystem allows access to any file on the MediaRich server, including files available via UNC paths (files on the network).

Because this filesystem allows such broad access to resources, it is read-only and disabled by default.

To enable the filesystem, add the following line to the `local.properties` file:

```
filesystem.file.enabled=true
```

To make the filesystem writable, add the following line:

```
filesystem.file.writeable=true
```

Examples

```
var image = new Media();
      image.load(name @ "file:C:/Documents and Settings/All
Users/Documents/My Pictures/Sample Pictures/Sunset.jpg");

var image2 = new Media();
      image2.load(name @
"file://eqfileserv/home/eng/testbed/images/tif/32bit.tif");
```

HTTP Support Using the FSNet Plug-in

The FSNet plug-in can implement HTTP and FTP access via standard URLs by defining virtual filesystems named “http” and “ftp”.

FSNet and MediaRich Virtual Filesystems

MediaRich accesses files by defining a number of virtual filesystems. A virtual file system indicates the root of a file tree located somewhere on either the local filesystem or on the network. When referencing a file via a virtual file path, the virtual filesystem on which the file is located is specified by prepending the virtual filesystem name followed by a colon onto the path to that file.

The FSNet plug-in can implement HTTP and FTP access via standard URLs by defining virtual filesystems named “http” and “ftp”. Since normal HTTP and FTP URLs consist of these names followed by a colon and then followed by a file path, normal HTTP and FTP URLs are valid MediaRich virtual file paths.

In addition to the default “http” and “ftp” filesystems, it is possible to set up other filesystems that refer to resources on HTTP and FTP servers. These URLs would look the same as the standard URLs except that the “http” or “ftp” keys at the beginning of the URLs would be some other name to specify that an alternate filesystem is being accessed. For more information about defining additional HTTP and FTP filesystem, see “Defining Additional FSNet Virtual Filesystems” on page 19.

Enabling Standard HTTP and FTP URL Access

To allow standard HTTP URLs to be passed to MediaRich as file paths, add the following line to MediaRich's `local.properties` file:

```
filesystem.fsnet.http.Specifier=http
```

To allow standard FTP URLs to be passed to MediaRich as file paths, add the following line to the `local.properties` file:

```
filesystem.fsnet.ftp.Specifier=ftp
filesystem.fsnet.ftp.Ftp=1
```

HTTP and FTP URLs

There are a number of MediaScript methods, such as the Media object's `load()` and `save()` methods, that accept file paths as parameters. The paths passed via these parameters will often be paths to files on the local filesystem, but can also refer to resources on a network via the HTTP and FTP protocols.

By enabling standard HTTP and FTP URLs as described in the previous section, references to files on HTTP and FTP servers take the same form as standard URLs used to access those files via a web browser. All of the information normally encoded in HTTP and FTP URLs can be passed to MediaRich. MediaRich will use the supplied information to connect to the specified network resource.

A fully specified HTTP or FTP URL looks like the following:

```
(http|ftp)://<username>:<password>@<server name>:<port>/
<path to resource>
```

The <username>, <password>, and <port> portions of the URL are optional. Here are some examples of well formed URLs:

```
http://www.eq.com/images/eq_bw.gif
http://joeblow:abcdefg@acomputer/myimages/blah.jpg
ftp://ftpserver:1234/public/images/camera.tif
```

FSNet Properties

A number of properties can be set in the local.properties file to influence the behavior of HTTP and/or FTP requests. These properties can be specified such that they affect all FSNet filesystems or so that they affect only a single filesystem.

To set a property that affects all FSNet filesystems, specify that property as follows:

```
filesystem.fsnet.<property name>
```

To set a property to only affect a single virtual filesystem, specify the property as follows:

```
filesystem.fsnet.<virtual filesystem name>.<property name>
```

The following example sets up a proxy host for all FSNet filesystems:

```
filesystem.fsnet.ProxyHost=ourproxyhost:3322
```

The following example sets the user name and password for just FTP access:

```
filesystem.fsnet.ftp.UserNameAndPassword=joeblow:tokyo
```

The specific properties that affect FSNet's operation are described in the following sub-sections. As you read this information, remember that each of these properties can be specified such that they affect either a single virtual filesystem ("http" or "ftp"), or all FSNet filesystems ("http" AND "ftp").

Authentication

Authentication is performed by supplying a user name and password to HTTP and FTP requests. These values can be supplied in the URL, as described in the previous section. They can also be set globally so that all HTTP and/or FTP requests use the same user name and password. This is done using the `UserNameAndPassword` property.

The following are two examples, one that sets a user name and password for all FSNet filesystems, and another that sets them only for FTP access:

```
filesystem.fsnet.UserNameAndPassword=joeblow:tokyo
filesystem.fsnet.ftp.UserNameAndPassword=tomjones:apassword
```

File Caching

The process of reading a file over a network can be time consuming. For this reason, files read via the FTP and HTTP protocols are stored in a local file cache. Subsequent accesses to the same file that occur shortly after the file was last downloaded are served from the cache rather than reread from the network.

The `RefreshInterval` property determines how HTTP and FTP files are cached. This value is interpreted as a time interval in seconds and indicates how often MediaRich should check with a HTTP or FTP server to determine if a newer version of a file exists. The default value of this property is 900, or 15 minutes. If the value of the `RefreshInterval` property is set to a negative value, then caching is disabled and every HTTP or FTP request retrieves a file from the network.

Unless caching has been disabled, caching behavior works according to the following sequence of events whenever a file is requested via HTTP or FTP:

MediaRich first looks to determine if the same file exists in the local cache.

- If the file does not exist in the cache, MediaRich fetches the file.
- If it does exist, MediaRich computes the age of that file (how long it has been since that file was last updated).
  - If the age of the file is less than that specified using the `RefreshInterval` property, MediaRich uses the cached file.
  - If the age of the file is greater than that specified using the `RefreshInterval` property, MediaRich contacts the HTTP or FTP server to determine whether a newer version of the file exists.
    - If a newer version does exist, MediaRich downloads a new copy of the file to the cache.
    - If a newer version does not exist, the age of the file in the cache is simply reset to 0. In all cases, the cached file is eventually returned to the caller.

This example disables file caching for FTP accesses:

```
filesystem.fsnet.ftp.RefreshInterval=-1
```

And this example refreshes interval for both HTTP and FTP accesses to five minutes:

```
filesystem.fsnet.RefreshInterval=300
```

Proxy Server Support

To route HTTP and FTP file requests through a proxy server, use the ProxyHost property. A proxy host specification consists of a host name and port number, separated by a colon. The following is an example that designates that FTP and HTTP requests should be routed to port #1133 of the server named "`ourproxyserver.ourcompany`":

```
filesystem.fsnet.ProxyHost=ourproxyserver.ourcompany:1133
```

Defining Additional FSNet Virtual Filesystems

Any number of virtual filesystems can be defined in addition to standard "http" and "ftp" filesystems. The only reason to define additional HTTP or FTP filesystems would be to associate different property settings with the different filesystems. For example, if you needed to access network files via two different proxy servers, you would need two different filesystems, as the proxy server can not be specified in a virtual file path (in a URL). Another reason to define additional filesystem would be to associate different user names and passwords with each filesystem, or to differ the caching behavior between filesystems.

To define a HTTP filesystem, only the `Specifier` property is required. The following is an example that defines a new HTTP filesystem named "media" and sets the default user info for that filesystem:

```
filesystem.fsnet.media.Specifier=media
filesystem.fsnet.media.UserNameAndPassword=joeblow:tokyo
```

A file path to a resource on this filesystem would look like this:

```
media://www.apple.com/images/applelogo.jpg
```

To define a FTP filesystem, the `Ftp` property must be specified to indicate that the filesystem will utilize the FTP protocol. The following is an example that defines a new FTP filesystem named "bar" that has caching disabled:

```
filesystem.fsnet.bar.Specifier=bar
filesystem.fsnet.bar.Ftp=1
filesystem.fsnet.bar.RefreshInterval=-1
```

The name of the filesystem, as specified after the "fsnet" portion of property name, need not match the value of the Specifier property. The filesystem name is used to refer to the filesystem in the properties file. The Specifier defines what word appears at the front of a virtual file path to indicate that filesystem. By convention, and for clarity, these two values should always be the same.

Configuring the FSNet File Systems

The FTP and HTTP file systems are disabled by default. To configure the file systems, you need to add entries to the `local.properties` file. The basic entries required to enable both file systems are:

```
filesystem.fsnet.http.Specifier=http
filesystem.fsnet.ftp.Specifier=ftp
filesystem.fsnet.ftp.FTP=1
```

The following table lists the properties you can configure for either FTP or HTTP:

Property	Usage
`Disabled`	Set to 1 to disable the file system, 0 or no entry to enable it
`Specifier`	This is the path specifier that denotes this file system.
`UserName`	Sets the default user name.
`Password`	Sets the default password.
`UserNameAndPassword`	Sets the default username and password (separated by ":").
`FTP`	Set to 1 to use FTP protocol, 0 or no entry to use HTTP.
`ProxyHost`	Set as proxyServerNameOrIP[:port] to configure a HTTP proxy server (HTTP only).
`ResponseTimeout`	Specifies the amount of time to wait for a response from a server before giving up.
`LockTryDelay`	Specifies the amount of time to wait (in seconds) when a cache file is locked.
`MaxTries`	Specifies how many times to retry a locked cache file before giving up.
`BreakLockOnFailure`	Set to 1 break a lock on a cache file, if the lock fails to become available.
`RefreshInterval`	Specifies how often (in seconds) to check the web for a newer image.

Chapter 3
MediaScript Objects and Methods

MediaScript includes a number of built-in objects that you can use to customize your MediaRich requests and responses.

Chapter Summary

Error Handling

Most MediaScript functions indicate an error condition by throwing an exception rather than returning an error code. Exceptions can be trapped and handled using ECMAScript's standard try/catch/finally mechanism.

For example, the Media object's `load()` method throws an exception if the file to be loaded is not found. To trap this exception, you would write something like the following:

```
try
{
    img.load(name @ "missingFile.tga");
}
catch (e)
{
    // Here you can recover from the error.  Possible
    // actions include loading a default image,
    // logging an error, or returning a 404.
}
```

If an exception is thrown while executing your script and no catch block traps it, the script will terminate immediately. By default the error is logged to the `ScriptErrors.log` file and returned to the client as an error response. You can disable logging by setting the `ScriptErrorLogging` property to false in the `global.properties` configuration file. For HTTP clients, you can disable the returning of the error response as HTML by setting the `ReturnHtmlErrors` property to false in global.properties. If error HTML is disabled MediaRich will respond with a 500 Internal Server Error status code.

File Object

The File object provides access to an external file.

Object Methods

- `new File() constructor`
- `clear()`
- `close()`
- `copy()`
- `exists()`
- `getFileName()`
- `getFilePath()`
- `getLastModified()`
- `getLastAccessed()`
- `getParentPath()`
- `getSize()`
- `isDirectory()`
- `isFile()`

- length()
- list()
- mkdir()
- read()
- readNextLine()
- remove()
- rename()
- rmdir()
- write()

new File()

The File object needs to be constructed using the `new File ()` constructor.

Syntax

```
var Test = new File(
       <"filename">
       );
```

Parameters

`filename` - specifies a string containing the filename and path with which the object is associated. The default is an empty string. The string must be in quotes. If the string does not specify a file system the default is the `write` file system. See "File Systems" on page 14 for more information.

clear()

Clears the contents of the file pointed to by the file object, if the file already exists.

Syntax

```
<object name>.clear();
```

Parameters

This function takes no parameters.

close()

Closes the file pointer and flushes any output immediately.

Syntax

```
<object name>.close();
```

Parameters

This function takes no parameters.

copy()

Copies the File object to a new filename. The object's original filename is unchanged.

Syntax

```
<object name>.copy(
        <"source">
        );
```

Parameters

`source` - specifies a string containing the new filename. The string must be in quotes. If the string does not specify a file system the default is the `write` file system. See "File Systems" on page 14 for more information.

exists()

Returns true if the File object points to an existing file or directory. Otherwise it returns false.

Parameters

This function takes no parameters.

getFileName()

Returns the filename portion of the object's path.

Syntax

```
<object name>.getFileName();
```

If the object's full file path is `C:\Program Files\Equilibrium\MediaRich\Shared\Originals\Media\camera.png`, `getFileName()` returns "camera.png".

Parameters

This function takes no parameters.

getFilePath()

Returns the full file path for the object.

Syntax

```
<object name>.getFilePath();
```

If the object's full file path is:

```
C:\Program
Files\Equilibrium\MediaRich\Shared\Originals\Media\camera.png
```

`getFilePath()` returns:

```
"C:\Program
Files\Equilibrium\MediaRich\Shared\Originals\Media\camera.png"
```

Parameters

This function takes no parameters.

getLastModified()

Returns the last modified date in seconds since midnight January 1, 1970. If the file does not exist or cannot be accessed, the function returns 0.

Syntax

```
<object name>.GetLastModified();
```

Parameters

This function takes no parameters

getLastAccessed()

Returns the last accessed date in seconds since midnight January 1,1970. If the file does not exist or cannot be accessed, the function returns 0.

Syntax:

```
<object name>.GetLastAccessed();
```

Parameters

This function takes no parameters

getParentPath()

Returns the parent path for the object.

Syntax

```
<object name>.getParentPath();
```

If the object's full file path is `C:\Program Files\Equilibrium\MediaRich\Shared\Originals\Media\camera.png`, `getParentPath()` returns "`C:\Program Files\Equilibrium\MediaRich\Shared\Originals\Media\`".

Parameters

This function takes no parameters.

getSize()

Returns the file size in bytes. If file does not exist, cannot be accessed, or is empty, the function returns 0.

Syntax:

```
<object name>.GetSize();
```

Parameters

This function takes no parameters

isDirectory()

Returns `true` if the File object points to a directory, otherwise returns `false`.

Syntax

```
<object name>.isDirectory();
```

Parameters

This function takes no parameters.

isFile()

Returns `true` if the File object points to a regular file, otherwise returns `false`.

Syntax

```
<object name>.isFile();
```

Parameters

This function takes no parameters.

length()

Returns an integer containing the number of text lines in the file.

Syntax

```
<object name>.length();
```

Parameters

This function takes no parameters.

list()

Returns an array of File objects representing the directory entries for the named File object. If the named File object does not represent a directory (or that directory is empty), then `list()` returns an empty array.

Syntax

```
<object name>.list();
```

Parameters

This function takes no parameters.

mkdir()

Creates the directory (and any non-existent parent directories) specified by the current File object.

Syntax

```
<object name>.mkdir();
```

Parameters

This function takes no parameters.

read()

Reads the specified line number from the file and returns it as a string.

NOTE: *Line numbers start at zero, not at one.*

Syntax

```
<object name>.read(
        <index>
        );
```

Parameters

`index` - specifies the line number and can range from 0 to 16,777,215.

readNextLine()

Reads the next line from the file pointed at by the File object, and returns a string containing the text in that line. Returns `undefined` at the end of the file.

Syntax

```
<object name>.readNextLine();
```

Parameters

This function takes no parameters.

remove()

Deletes the filename for the File object.

Syntax

```
<object name>.remove();
```

Parameters

This function takes no parameters.

rename()

Renames the File object, changing the name property of the object to the new name.

Syntax

```
<object name>.rename(
        <"newname">
        );
```

Parameters

`newname` - a string containing the new name property. The string must be in quotes. If the string does not specify a file system the default is the `write` file system. See "File Systems" on page 14 for more information.

rmdir()

Removes the specified directory and returns true if the directory could be removed and false if it could not.

Syntax

```
var success = <object name>.rmdir(
        <"recurse">
        );
```

Parameters

`recurse` - controls whether the contents of the specified directory are deleted. If this is set to true, the directory and all of its content is removed. If set to false, the directory is only deleted if it is empty.

write()

Writes a string to the end of the file.

Syntax

```
<object name>.write(
        <"string">
        );
```

Parameters

`string` - specifies the string to write. The string must be in quotes.

Global Request and Response Objects

For every request, MediaScript creates a global HTTPRequest object named "`req`" and a global HTTPResponse object named "`resp`". You can use the request object to get request parameters, HTTP headers, and information about the MRL. The request object interface is described in detail in the HTTPRequest reference section. You can use the response object to set the response contents, HTTP response headers, and status code. See "HTTP Response Methods" on page 36 for more details.

Setting the Response Contents

There are several ways to set the response contents. The simplest is to use the Media object's `save()` method without a name parameter. For example:

```
img.save(type @ "jpeg");
```

When the script terminates, the contents of the img object are saved to the cache so that subsequent requests can be returned immediately without having to re-execute the script.

The response contents can also be set explicitly using the response object's `setObject()` method. For example, the previous example could be rewritten as:

```
resp.setObject(img, RespType.Cached, {type: "jpeg"});
```

The response object can be any of the following types: Media, XmlDocument, or TextResponse. For more information about the `setObject()` method, see “setObject()” on page 37.

Finally, the response contents can be set directly to the path of an existing file using the response object's `setPath()` method. For example, to return a PDF file in the media directory named response.pdf, you would write:

```
resp.setPath("response.pdf");
```

By default the file will be copied to the cache. You can also stream the contents back to the client or just return the file path by setting the response type to one of the values listed in the following section.

Response Types

There are three types of responses:

- `RespType.Cached` - This is the default response type. The response object or file is saved to the cache so that subsequent requests with the same parameters will be returned immediately from the cache. The response in this case is the full path of the new cached file.
- `RespType.Streamed` - The response contents are set to the contents of the response object or file. Since nothing is saved in the cache, subsequent requests will re-execute the script.
- `RespType.Path` - This type applies to response files only. The response is set to the full path of the file. Again, nothing is saved in the cache so subsequent request will re-execute the script.

The Response Object does not need to be constructed. A static global instance of each is created for each MediaScript execution context. The `resp` object allows the user to set the Media response.

HTTP Request Object Properties

This object has no properties.

HTTP Request Methods

The `req` object includes the following methods:

- `getParameter()`
- `getParameterNames()`
- `getHeader()`
- `getFileParamNames()`
- `getFileParamPath()`
- `getHeaderNames()`
- `getRequestURL()`
- `getQueryString()`
- `getScriptPath()`

- `getJobId()`
- `getBatchId()`
- `getJobTempDir()`
- `getBatchTempDir()`

getParameter()

Returns the MRL parameter specified by name, or null if no such parameter exists.

Syntax

```
var paramValue = req.getParameter(
       <name>
       );
```

Parameters

`name` - specifies the name of the MRL parameter to retrieve.

getParameterNames()

Returns an array of the names of all parameters specified on the MRL.

Syntax

```
var nameArray = req.getParameterNames();
```

Parameters

This function takes no parameters.

getHeader()

Returns the value for the given HTTP header name.

NOTE: *getHeader is both a request method where the request headers are returned, and a response method, where the response headers are returned.*

Syntax

```
req.getHeader(
       <name>
       );
```

Parameters

`name` - specifies the header name.

Example

```
function main() {
        var respText = new TextResponse(TextResponse.TypePlain);
        var headers = req.getHeaderNames();
        for (var i = 0; i < headers.length; ++i)
        {
              respText.append(headers[ i]  + ": " +
req.getHeader(headers[ i] ) + "\n");
        }
        resp.setObject(respText, RespType.Streamed);
}
```

Sample Output

```
accept: */*
accept-encoding: gzip, deflate
accept-language: en-us
connection: Keep-Alive
host: localhost
user-agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;
Q312461)
```

getFileParamNames()

Returns an array containing the names of all file parameters. File parameters describe files that have been sent to the Media Generator through the .NET or Java APIs for processing. The names returned by this method may be used in conjunction with `getFileParamPath()` to locate the files reference by the file parameters. These names are specified when the file is added as a parameter to a request using the .NET of Java APIs. Please refer to the Java or .NET API documentation for more details.

Syntax

```
var fileNameList = req.getFileParamNames();
```

Parameters

This function takes no parameters.

getFileParamPath()

Returns the path to the file associated with the specified file parameter name.

Syntax

```
var myImageFile = req.getFileParamPath("paramName");
```

Parameters

`paramName` - the name specified for the file parameter when it was set with the request.

Example

Suppose that a file parameter is passed to a script with a parameter name of "testFile". This file can be accessed within a script as follows:

```
var path = req.getFileParamPath("testFile");
var m = new Media();
m.load(name @ path);
```

getHeaderNames()

Returns an array of all HTTP header names for the current request.

Syntax

```
req.getHeaderNames();
```

Parameters

This function has no parameters.

Example

```
function main() {
	var respText = new TextResponse(TextResponse.TypePlain);
	var headers = req.getHeaderNames();
	for (var i = 0; i < headers.length; ++i)
	{
		respText.append(headers[ i]  + ": " +
req.getHeader(headers[ i] ) + "\n");
	}
	resp.setObject(respText, RespType.Streamed);
}
```

Sample Output

```
accept: */*
accept-encoding: gzip, deflate
accept-language: en-us
connection: Keep-Alive
host: localhost
user-agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;
Q312461)
```

getRequestURL()

Returns the full MRL that originated the request.

Syntax

```
var url = req.getRequestURL();
```

Parameters

This function has no parameters.

getQueryString()

Returns the query string portion of the MRL that originated the request (everything after the "?").

Syntax

```
var queryString = req.getQueryString();
```

Parameters

This function has no parameters.

getScriptPath()

Returns a string containing just the query component of the URL used to generate the executing request.

Syntax

```
req.getScriptPath();
```

Parameters

This function has no parameters.

Example

If the original URL is, `http://MRserver/mgen/fotophix/photochange.ms?args=%22pic3.jpg%22&is=80,50&p=4:1`, then `req.ScriptPath()` returns "`/fotophix/photochange.ms`".

getJobId()

Returns an integer identifying the current job. Note that this identifier is unique only to a specific MediaGenerator. It is reset to start at 0 when the MediaGenerator is started and is incremented by 1 for each request processed. See the .NET or Java API documentation for a description of the MediaRich batch interface.

Syntax

```
var id = req.getJobId();
```

Parameters

This function has no parameters.

getBatchId()

Returns an integer identifying the "Batch". A batch is a collection of jobs sent to the Media Generator through the .NET of Java APIs. Note that this identifier is unique only to a specific MediaGenerator. It is reset to start at 0 when the MediaGenerator is started and is incremented by 1 for each batch request. See the .NET or Java API documentation for a description of the MediaRich batch interface. For non-batch requests, this identifier is always 0.

Syntax

```
var batchId = req.getBatchId();
```

Parameters

This function has no parameters.

getJobTempDir()

Returns the path to a local temporary directory for use by the current job. This directory is created at the start of each request and deleted when the request completes.

Syntax

```
var tmpDir = req.getJobTempDir();
```

Parameters

This function has no parameters.

getBatchTempDir()

Returns the path to a local temporary directory that is shared by all jobs within a given batch. This directory may be used to share state between different jobs in a given batch. For a description of the MediaRich batch interface, see the .NET or Java API online documentation available in the SDK folder.

Syntax

```
var tmpDir = req.getBatchTempDir();
```

Parameters

This function takes no parameters.

getPath()

Returns the path previously set using setPath().

Syntax

```
resp.getPath();
```

Parameters

This function takes no parameters.

HTTP Response Methods

The `resp` object includes the following methods:

- `setObject()`
- `getObject()`
- `setPath()`
- `getPath()`
- `setMedia()`
- getMedia()
- `setResponseType()`
- `getResponseType()`
- `setSaveParameters()`

- `getSaveParameters()`
- `setMimeType()`
- `getMimeType()`
- `setHeader()`
- `getHeader()`
- `setStatusCode()`
- `getStatusCode()`
- `write()`
- `writeLine()`

setObject()

Sets the response object. Optional parameters can set the response type and save parameters for the specified object.

Syntax

```
resp.setObject(
        [obj]
        [respType]
        [saveParams{}]
);
```

Parameters

obj - specifies the response object.

RespType - optional parameter that sets the response type:

- RespType.Cached (the default) saves the response object or file to the MediaResults cache directory so that future requests are returned directly by the filter.
- RespType.Streamed bypasses the cache and returns the response data directly to the filter.
- RespType.Path returns the full native path of a file to the filter but does not copy the file to the cache.

For more information, see setResponseType().

saveParams - optional parameter that saves response parameters as an object. The object must be pre-existing or a temporary object using the {} syntax.

Example

```
var img = new Media();
img.load(name @ "foo.jpg");
resp.setObject(img, RespType.Streamed, {type: "png"});
```

getObject()

Returns the object previously set by either `setObject()`, `setMedia()`, or a Media::`save()` call with no name.

NOTE: *If no response object is set, the function returns null.*

Syntax

```
resp.getObject();
```

Parameters

This function has no parameters.

setPath()

Sets the response file path. An optional parameter can set the response type.

Syntax

```
resp.setPath(filePath, RespType);
```

Parameters

`filePath` - sets the response file path. If the path does not specify a file system the default is the `read` file system. See "File Systems" on page 14 for more information.

`RespType` - optional parameter that sets the response type:

- `RespType.Cached` (the default) saves the response object or file to the MediaResults cache directory so that future requests are returned directly by the filter.
- `RespType.Streamed` bypasses the cache and returns the response data directly to the filter.
- `RespType.Path` returns the full native path of a file to the filter but does not copy the file to the cache.

For more information, see "setResponseType()" on page 39.

Example

```
var img = new Media(); img.load(name @ "foo.jpg");
img.save(name @ "/bar.gif", type @ "gif");
resp.setPath("/bar.gif", RespType.Path);
```

getPath()

Returns the path of the current response Media object.

Parameters

This function takes the name of a Media object as its only parameter.

Syntax

```
resp.getPath(
       <Media object>
       );
```

setMedia()

Sets the response Media object to the specified object.

Parameters

This function takes the name of a Media object as its only parameter.

Syntax

```
resp.setMedia(
        <Media object>
        );
```

getMedia()

Returns the current response Media object.

Syntax

```
resp.getMedia();
```

Parameters

This function has no parameters.

setResponseType()

Sets the response type.

Syntax

```
resp.setResponseType(
        [ RespType]
        );
```

Parameters

RespType - sets the response type:

- `RespType.Cached` saves the response object or file to the MediaResults cache directory so that future requests are returned directly by the filter.
- `RespType.Streamed` bypasses the cache and returns the response data directly to the filter.
- `RespType.Path` (the default) returns the full native path of a file to the filter but does not copy the file to the cache.

Example

```
var img = new Media(); img.load(name @ "foo.jpg");
img.save(type @ "gif");
resp.setResponseType(RespType.Streamed);
```

getResponseType()

Returns the response type previously set by setPath(), setObject(), or setResponseType().

NOTE: *If no response path is set, the function returns null.*

Syntax

```
var respType = resp.getResponseType();
```

Parameters

This function has no parameters.

setSaveParameters()

Sets the response save parameters for the response media object.

Syntax

```
resp.setSaveParameters(
       <save parameters>
);
```

Parameters

`save parameters` - specified as either an object, or in the standard syntax (for example, `type @ "jpeg"`). These parameters are passed directly to the Media object `save` method. For a description of these parameters, see "save()" on page 124.

getSaveParameters()

Returns the current save parameters for the response Media object.

Syntax

```
resp.getSaveParameters();
```

Parameters

This function has no parameters.

setMimeType()

Sets the response MIME type.

Syntax

```
resp.setMimeType("text/xml");
```

Parameters

mimeType - specifies the type of response. The default type depends on the response type:

- For files, it is determined automatically based on the file extension.
- For object responses, it is based on the value returned by the `_MR_save()` method of the object.

getMimeType()

Returns the response MIME type or, if not set, undefined.

Syntax

```
var mimeType = req.getMimeType();
```

Parameters

This function has no parameters.

setHeader()

Sets a HTTP response header with the given name/value pair.

Syntax

```
resp.setHeader("name", "value");
```

Parameters

`name` - the string name indicating the HTTP header

`value` - the string value of the HTTP header.

getHeader()

Returns the value for the given HTTP header name.

Syntax

```
req.getHeader(
        <name>
        );
```

Parameters

The only parameter is the specified header name.

Example

```
function main() {
        var respText = new TextResponse(TextResponse.TypePlain);
        var headers = req.getHeaderNames();
        for (var i = 0; i < headers.length; ++i)
        {
              respText.append(headers[ i]  + ": " +
req.getHeader(headers[ i] ) + "\n");
        }
        resp.setObject(respText, RespType.Streamed);
}
```

Sample output

```
accept: */*
accept-encoding: gzip, deflate
accept-language: en-us
connection: Keep-Alive
host: localhost
user-agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;
Q312461)
```

setStatusCode()

Sets the HTTP response status code. The default is STATUS_OK (200).

Syntax

```
resp.setStatusCode("statusCode");
```

Parameters

`statusCode` - an integer value for the HTTP response status. The default is STATUS_OK (200).

getStatusCode()

Returns the response status code.

Syntax

```
var code = resp.getStatusCode();
```

Parameters

This function has no parameters.

write()

Appends the specified text to the response object. When you use this method, you should set the mime type to the type of text written (such as text/plain, text/xml, etc.).

Syntax

```
resp.write("This is the response");
resp.setMimeType("text/plain");
```

Parameters

The only parameter is the specified text string.

writeLine()

Appends the specified text to the response object and adds a new line. When you use this method, you should set the mime type to the type of text written (such as text/plain, text/xml, etc.).

Syntax

```
resp.writeLine("This is the response with a newline");
resp.setMimeType("text/plain");
```

Parameters

The only parameter is the specified text string.

Media Object

The Media object implements the many image processing features provided by MediaRich. A typical script creates one or more Media objects by either loading image data from a file or using the Media drawing methods.

Class Methods

There are static methods on the Media object. They have been added to the Media class, and include:

- `getFileInfo()`
- `getFileFormats()`
- `getExtensionFromType()`
- `getTypeFromExtension()`

getFileInfo()

The method returns data about the image or images contained in the specified file. It returns as much information as can be ascertained without "considerable processing time." It is left for the implementor of each file format handler to determine what "considerable processing time" means, but what is normally returned is that information that can be obtained by reading a small portion (1-2K) of the file. The data is returned as a JavaScript object, where each value returned is a property of that object.

The information returned by this method will vary from one file type to another. The only value that is guaranteed to be returned is the "Type" value. The "Type" value is the name of the file type of the file (such as tiff, jpeg, etc.)

The following values are returned by most file types (current exceptions are ".ai", ".pct", ".ps", ".eps" and ".pdf" files):

- Width - the width of the image(s) contained in the file
- Height - the height of the image(s) contained in the file
- Format - the pixel format of the image(s) contained in the file

Other values that are returned by some file types are:

- XDpi - the horizontal resolution of the image(s)
- YDpi - the vertical resolution of the image(s)
- Frames - the number of frames in the file
- Layers - the number of layers in the file

Syntax

```
var fileInfo = Media.getFileInfo(name @ "tif/32bit.tif");
```

Parameters

The only parameter is the specified file name.

Example

```
var fileInfo = Media.getFileInfo(name @ "tif/32bit.tif");
for (key in fileInfo)
{
       print(key + ":" + fileInfo[ key]  + "\n");
}
```

Individual properties can be accessed like this:

```
var fileInfo = Media.getFileInfo(name @ "tif/32bit.tif");
var width = fileInfo[ "Width"];
```

getFileFormats()

Use this method to get a description of the available file formats. It returns an array of objects describing the available file formats.

Each element of the array contains the following fields:

- type: The file format type.
- extensions: A comma-delimited list of file extensions for the format.
- flags: A bitwise-OR (|) of one or more of the following flags:

`Media.FormatLoad`	-> Indicates that the format is loadable.
`Media.FormatSave`	-> Indicates that the format is saveable.
`Media.FormatCmykSave`	-> Indicates that the format can save CMYK files.
`FormatExifLoad`	-> Use if the format supports loading Exif metadata.
`FormatExifSave`	-> Use if the format supports saving Exif metadata.
`FormatIPTCLoad`	-> Use if the format supports loading IPTC metadata.
`FormatIPTCSave`	-> Use if the format supports saving IPTC metadata.
`FormatXMPLoad`	-> Use if the format supports loading XMP metadata.
`FormatXMPSave`	-> Use if the format supports saving XMP metadata.

Syntax

```
var fileFormats = Media.getFileFormats();
```

Parameters

This function takes no parameters.

Example

This example returns a text file describing all of the available file formats.

```
#include "sys:/TextResponse.ms"

function main()
{
    var foo = Media.getFileFormats();
    var txt = new TextResponse();
    for (var i = 0; i < foo.length; ++i)
    {
        txt.append(foo[i].type + ":\n");
        txt.append("  exts: " + foo[i].extensions + "\n");
        txt.append("  flags: ");
        if (foo[i].flags & Media.FormatLoad)
            txt.append("load ");
        if (foo[i].flags & Media.FormatSave)
            txt.append("save ");
        if (foo[i].flags & Media.FormatCmykSave)
            txt.append("cmyk");
        txt.append("\n");
    }
    resp.setObject(txt, RespType.Streamed);
}
```

getExtensionFromType()

Returns the file system extension (such as “jpg”) for the given the media type.

Syntax

```
var extension = Media.getExtensionFromType(
      <"type">
      );
```

Parameters

type - the media type (such as “jpeg”, “tiff”, etc.).

getTypeFromExtension()

Returns the media type for the given the extension.

Syntax

```
var type = Media.getTypeFromExtension(
      <"extension">
      );
```

Parameters

`extension` - the file system extension whose type is desired (such as "psd").

Object Methods

The Media object includes the following object methods:

- new Media() constructor
- `addArgument()`
- `adjustHsb()`
- `adjustRgb()`
- `arc()`
- `blur()`
- `blurBlur()`
- `blurGaussianBlur()`
- `blurMoreBlur()`
- `blurMotionBlur()`
- `clone()`
- `collapse()`
- `colorCorrect()`
- `colorize()`
- `colorFromImage()`
- `colorToImage()`
- `composite()`
- `convert()`
- `crop()`
- `discard()`
- `drawText()`
- `dropShadow()`
- `ellipse()`
- `embeddedProfile()`
- `equalize()`
- `exportChannel()`
- `fixAlpha()`
- `flip()`
- `frameAdd()`
- `getBitsPerSample()`
- `getBytesPerPixel()`
- `getFrame()`
- `getFrameCount()`
- `getHeight()`

- getImageFormat()
- getInfo()
- getLayer()
- getLayerBlend()
- getLayerCount()
- getLayerEnabled()
- getLayerHandleX()
- getLayerHandleY()
- getLayerIndex()
- getLayerName()
- getLayerOpacity()
- getLayerX()
- getLayerY()
- getMetaData()
- getPalette()
- getPaletteSize()
- getPixel()
- getPixelFormat()
- getPixelTransparency()
- getPopularColor()
- getResVertical()
- getResHorizontal()
- getSamplesPerPixel()
- getWidth()
- getXmlInfo()
- glow()
- importChannel()
- infoText()
- line()
- load()
- loadAsRgb()
- makeCanvas()
- makeText()
- measureText()
- noiseAddNoise()
- otherHighPass()
- otherMaximum()
- otherMinimum()
- pixellateFragment()
- pixellateMosaic()

- polygon()
- quadWarp()
- reduce()
- rectangle()
- rotate()
- rotate3d()
- save()
- saveEmbeddedProfile()
- scale()
- selection()
- setColor()
- setFrame()
- setLayer()
- setLayerBlend()
- setLayerEnabled()
- setLayerHandleX()
- setLayerHandleY()
- setLayerOpacity()
- setLayerPixels()
- setLayerX()
- setLayerY()
- setMetadata()
- setPixel()
- setResolution()
- setSourceProfile()
- sharpenSharpen()
- sharpenSharpenMore()
- sharpenUnsharpMask()
- sizeText()
- stylizeDiffuse()
- stylizeEmboss()
- stylizeFindEdges()
- stylizeTraceContour()
- zoom()

new Media()

The Media object needs to be constructed using the `new Media()` constructor.

Syntax

```
var Test = new Media();
```

adjustHsb()

Alters the HSB levels of an image. It can be applied to images of all supported bit-depths.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see selection().*

Syntax

```
adjustHsb(
        [Hue @ <value ±255>]
        [Saturation @ <value ±255>]
        [Brightness @ <value ±255>]
        [UseHLS @ <value true,false>]
        );
```

Parameters

The default value for any parameter not specified is zero.

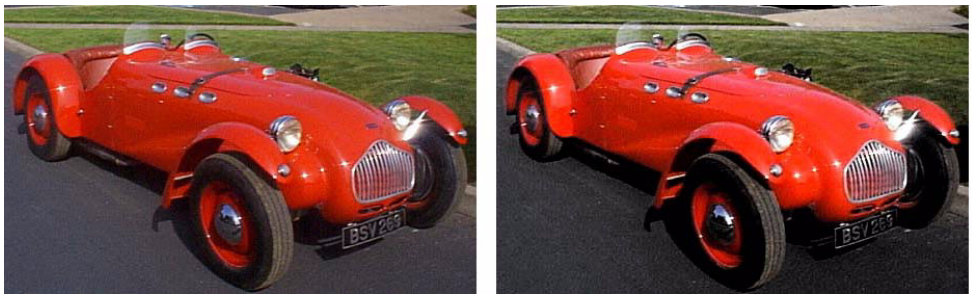
`Hue` - an angular color value, so the results from `hue @ -255` and `hue @ 255` are almost identical.

`Saturation` and `Brightness` - linear color values that set the base level for the saturation and brightness of the image.

`UseHLS` - if specified as "true", causes the adjustment to be performed in the HLS colorspace. In this case, the `Saturation` parameter is interpreted as lightness and the `Brightness` parameter is interpreted as Saturation.

Example

```
var image = new Media();
image.load(name @ "car.tga");
image.adjustHsb(hue @ 120, saturation @ 50, brightness @ 110);
image.save(type @ "jpeg");
```



addArgument()

Adds the specified name-value pair to the next Media object method call.

adjustRgb()

Alters the contrast, brightness, and color balance of an image.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see selection().*

Syntax

```
adjustRgb(
        [ Contrast @ <value ±255>]
        [ Brightness @ <value ±255>]
        [ Red @ <value ±255>]
        [ Green @ <value ±255>]
        [ Blue @ <value ±255>]
        [ NoClip @ <true, false>]
        [ Invert @ <true, false>]
        );
```

Parameters

The default value for any parameter not specified is `zero/false`.

`Contrast` - adjusts the overall contrast of the image.

`Brightness` - adjusts the overall brightness of the image.

`Red`, `Green`, and `Blue` - adjust the brightness of each of the three color channels individually.

`Noclip` - when specified, brightness adjustments will avoid maxing-out (either high or low) the image by reducing the contrast accordingly. A contrast offset can be used to override this process.

`Invert` - inverts the values of the three color channels. When mixed with any other settings in this command, all other calculations are performed first, then the inversion is applied as a last step.

Example

```
var image = new Media();
image.load(name @ "car.tga");
image.adjustRgb(red @ 120, blue @ 50, green @ 20, invert @ true);
image.save(type @ "jpeg");
```

arc()

Draws and positions an arc on the image based on the specified parameters. This method accepts all `composite()` parameters except `HandleX` and `HandleY`.

The foreground color may vary with this function, depending on the original Media object. If the object has a set foreground color, or it is set with the `setColor()` function, MediaRich uses the set color.

However, if the object has no set *foreground* color, MediaRich does the following:

- For objects with 256 colors or less, MediaRich uses the last color index
- For objects with 15-bit or greater resolution, MediaRich uses white

NOTE: *Using `arc()` to mask frames within a JavaScript `for` loop can result in initially poor anti-aliasing. To maintain optimal anti-aliasing, place the masking arc outside the loop.*

Syntax

```
arc(
        X @ <pixel>,
        Y @ <pixel>,
        Rx @ <value>,
        Ry @ <value>,
        Startangle @ <value -360..360>,
        Endangle @ <value -360..360>,
        [ Opacity @ <value 0..255>]
        [ Unlock @ <color in hexadecimal or rgb>]
        [ Color @ <color in hexadecimal or rgb>]
        [ Index @ <value 0..16777215>]
        [ Saturation @ <value 0..255>]
        [ PreserveAlpha @ <true, false>]
        [ Blend @ <"blend-type">]
        [ Width @ <value>]
        [ Smooth @ <true, false>]
        [ Fill @ <true, false>]
        [ Warpangles @ <true, false>]
        );
```

Parameters

The arc is created as a portion of a defined ellipse:

`X` - specifies (in pixels) the *x* axis coordinate for the center point of the ellipse from which the arc is derived. This parameter is required and has no default value.

`Y` - specifies (in pixels) the *y* axis coordinate for the center point of the ellipse from which the arc is derived. This parameter is required and has no default value.

`Rx` - specifies (in pixels) the radius of the ellipse (from which the arc is derived) on the *x* axis. This parameter is required and has no default value.

`Ry` - specifies (in pixels) the radius of the ellipse (from which the arc is derived) on the *y* axis. This parameter is required and has no default value.

`Startangle` - indicates the point of the ellipse (from which the arc is derived) where the arc begins. This parameter is required. There is no default value.

`Endangle` - indicates the point of the ellipse (from which the arc is derived) where the arc ends. This parameter is required. There is no default value.

`Opacity` - specifies opacity of the drawn object. The default value is 255 (completely solid).

`Unlock` - when set to true, causes the arc to display only where the specified color value appears in the current (background) image. The default is false.

`Color` - sets the color of the ellipse.

`Index` - if a color palette exists for the source image, use this parameter to set the color of the arc (as an alternative to the `Color` parameter).

NOTE: *You cannot enter values in both the `Color` and `Index` fields.*

`Saturation` - specifies a value for the weighting for the change in saturation for destination pixels. A value of 255 changes the saturation of pixels to the specified color. A value of 128 changes the saturation of a pixel to a mid-value between the pixel's current color and the specified color.

NOTE: *The `Saturation` parameter only functions when the `Blend` parameter is set to "colorize."*

`PreserveAlpha` - when set to true, preserves the alpha channel of the target image as the alpha channel of the resulting image. The default value is false.

`Blend` - specifies the type of blending used to combine the drawn object with the images. Blend options are: "Normal", "Darken", "Lighten", "Hue", "Saturation", "Color", "Luminosity", "Multiply", "Screen", "Dissolve", "Overlay," "HardLight", "SoftLight", "Difference", "Exclusion", "Dodge", "ColorBurn", "Under", "Colorize" (causes only the hue component of the source to be stamped down on the image), and "Prenormal".

NOTE: *"Burn" has been deprecated. "ColorBurn" results in the same blend.*

`Width` - specifies the thickness (in pixels) of the line that describes the arc. The default value is 1. However, if the `Fill` parameter is set to true, the `Width` parameter is ignored.

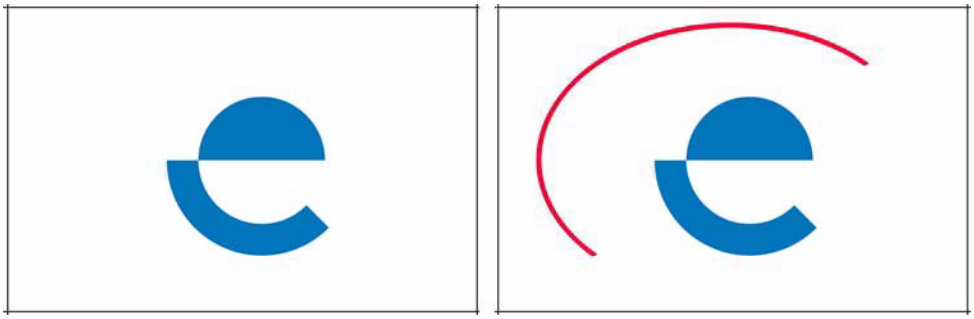
`Smooth` - when set to true, makes the edges of the arc smooth, preventing a pixellated effect. The default is false.

`Fill` - when set to true, fills in the arc with the color specified by the `Color` or `Index` parameter. The default value is false.

`Warpangles` - when set to true, warps the angles to match the ellipse. The default value is false.

Example

```
var image = new Media();
image.load(name @ "logobg.tga");
image.arc(X @ 185, Y @ 121, Rx @ 175, Ry @ 111, StartAngle @ -120,
EndAngle @ 60, Width @ 2, Smooth @ true, WarpAngles @ true);
image.save(type @ "jpeg");
```



blur()

Applies a simple blur filter on the image. For each pixel, all the pixels within the given radius are averaged and the result put in the destination image. This function fully supports CMYK.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see selection().*

Syntax

```
blur(
        Radius @ <value 0..30>
        );
```

Parameters

`Radius` - specifies the radius (in pixels) of the effect.

NOTE: *The "radius" is actually square, so a radius of two results in averaging over a 5x5 square centered on the given pixel.*

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.blur(radius @ 12);
image.save(type @ "jpeg");
```

blurBlur()

Applies a similar but milder blur effect as the blur() function.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
blurBlur();
```

Parameters

There are no parameters for this function.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.blurBlur();
image.save(type @ "jpeg");
```

blurGaussianBlur()

Applies a Gaussian blur effect to the image.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
blurGaussianBlur(
       [Radius @ <value 0.10..250>]
       );
```

Parameters

`Radius` - specifies the extent of the effect. The default is 1.00.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.blurGaussianBlur(Radius @ 5);
image.save(type @ "jpeg");
```

blurMoreBlur()

Applies a similar but stronger blur effect as the blurBlur() function.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
blurMoreBlur()
```

Parameters

There are no parameters for this function.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.blurMoreBlur();
image.save(type @ "jpeg");
```

blurMotionBlur()

Simulates the type of blur that results from motion (as in the photo of a tree photographed from a moving car).

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
blurMotionBlur(
        [Angle @ <value -360..360>]
        [Distance @ <value 1..250>]
        );
```

Parameters

`Angle` - specifies the direction of the blurring motion. The default is 0 (level, suggesting motion from left to right).

`Distance` - specifies the intensity or "motion speed" of the effect. The default is 10.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.blurMotionBlur(Angle @ 30, Distance @ 10);
image.save(type @ "jpeg");
```

clone()

Copies one Media object into another. After a Media object has been cloned, both the original and the copy can be modified independently, with changes to one object not affecting the other.

Syntax

```
<object name>.clone();
```

Parameters

This function has no parameters.

Example

```
var original = new Media();
original.load(name @ "weasel.tga");
original.scale(xs @ 250, constrain @ true);
var copy = original.clone()
...
```

collapse()

Collapses a multi-layer (Photoshop) file into a single layer. This function always results in a 32-bit image.

NOTE: *This function supports the CMYK color-space if all layers in the image are CMYK.*

Syntax

```
collapse(
      [ layers @ <"layer list">]
      [ PreserveAlpha @ <true, false>]
      [ IgnoreAlpha @ <true, false>]
      [ VisibleOnly @ <true, false>]
      [ likePS @ <true, false>]
      );
```

Parameters

`layers` - specifies the layers to collapse and the order in which to collapse them. The layer numbers begin at 0 (background) and go up. The default collapses all layers from bottom to top.

The layer list must be contained in quotes and consists of comma-separated entries. You can specify ranges ("0-2") or individual layers ("0,2"). If you specify the layers out of order, and they are composited accordingly.

NOTE: *When you specify a comma-separated list of layers, do not leave any spaces after the commas.*

If the Photoshop file has named layers, you can use the layer names (up to 31 characters) in place of layer numbers. You can also use the "*" as a wildcard when specifying layers. For example:

```
image.collapse(layers @ "B*")
```

This line of script collapses all layers whose names begin with "B" (such as Boy, Baseball, Ballcap, and so on). The layers command is case-sensitive, so the example line of script will not flip layers that begin with a lowercase "b".

`PreserveAlpha` - when set to true, preserves the alpha channel of the target image layers as the alpha channel of the resulting collapsed image. The default is false.

`IgnoreAlpha` - when set to true, the alpha channel information is ignored when collapsing the image layers. The default is false.

`VisibleOnly` - when set to "true", only the layers designated as visible are included in the collapsed image. The default is false.

`likePS` - if set to "true", performs the collapse like Photoshop: The default background color is white, and the alpha channel is determined by the first layer specified for the collapse. The default is false.

NOTE: *Because `likePS` uses a white background color, this parameter should be disabled for images that will later be used as a brush.*

Example

```
var image = new Media();
image.load(name @ "pasta.psd");
image.collapse(layers @ "0-2", likePS @ true);
image.save(type @ "jpeg");
```

colorCorrect()

Transforms an image from a source color space to a destination color space. MediaRich supports ICC profiles for the following formats: EPS, JPEG, PDF, PS, PSD, and TIFF.

Specifying ICC profiles:

The sourceProfile and destProfile parameters may be specified either as a filename or as an IccProfile object. By default, profiles are read from the "color:" file system which is defined by default as a combination of the `MediaRich/Shared/Originals/Profiles` directory and the system color profile directory if there is one. The `MediaRich/Shared/Originals/Profiles` directory is searched first.

In addition, the special profile names "rgb", and "cmyk" may be used to designate the default RGB and CMYK profiles specified in the global.properties file under the property keys "`ColorManager.DefaultRGBProfile`" and "`ColorManager.DefaultCMYKProfile`", respectively. You can change these to designate any default RGB or CMYK profiles you want.

NOTE: *If a Color Profile is associated with an RGB image, this is considered unnecessary data and by default the attached profiles will not be saved to RGB images.*

The MediaRich server's `local.properties` file may be modified to change the default profile directory. See the *MediaRich Administrator's Guide* for more information.

Syntax

```
colorCorrect(
        destProfile @ <"filename.icc">
        [ sourceProfile @ <"filename.icc">]
        [ intent @ <"rendering intent">
        [ overrideEmbedded @ <true, false>
        );
```

Parameters

`destProfile` - specifies the destination profile. After an image has been colorCorrected, this profile becomes the embedded profile for the image. The default location for destination profiles is: `MediaRich/Shared/Originals/Profiles`

NOTE: *You can modify the MediaRich server's local.properties file to change the default Profiles directory. See MediaRich Administrator's Guide for more information.*

`sourceProfile` - specifies the profile to be used as the source for the color transformation if the image has no embedded profile or if the `overrideEmbedded` parameter is set to true. Source profiles must also be located in the `/Profiles` directory. An ICC profile embedded in an image is used by default as the `sourceProfile`, unless the `overrideEmbedded` parameter is set to true.

`intent` - specifies how the transformation from source to destination color space is effected. The possible values for intent are:

- "`Perceptual`" - the default intent and works best with photographic images.
- "`RelativeColorimetric`" - corrects the image using the relative white points of the source and destination ICC profiles.

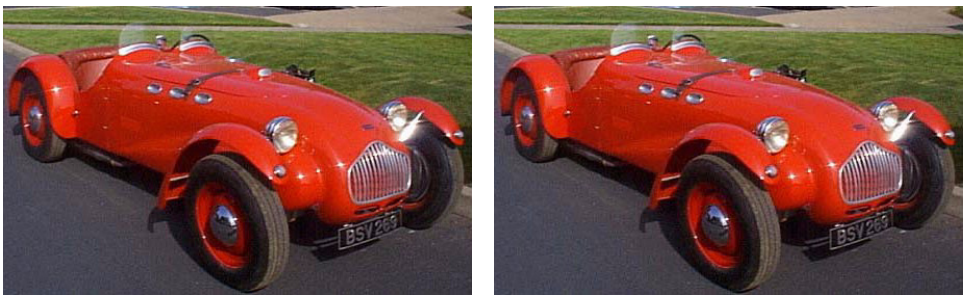
- "AbsoluteColorimetric" - corrects the image to the absolute white point specified in the destination ICC profile.
- "Saturation" - works best for line art and images that have large areas of one solid color.

NOTE: *For multi-layer images, profiles are associated only with the image and not with any of the layers; thus using `colorCorrect()` affects all layers.*

For more information, please see Appendix B, "MediaRich Color Management.".

Example

```
var image = new Media();
image.load(name @ "car.jpg");
image.colorCorrect(destProfile @ "sRGB.icc");
image.save(type @ "jpeg");
```



colorize()

Changes the hue of the pixels in the image to the specified color.

NOTE: *This function is "selection aware," and requires an active selection so that the system can apply `colorize()` based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
colorize(
        Color @ <color in hexadecimal or rgb>
        [ layers @ <"layer list">] // (PSD files only)
        );
```

Parameters

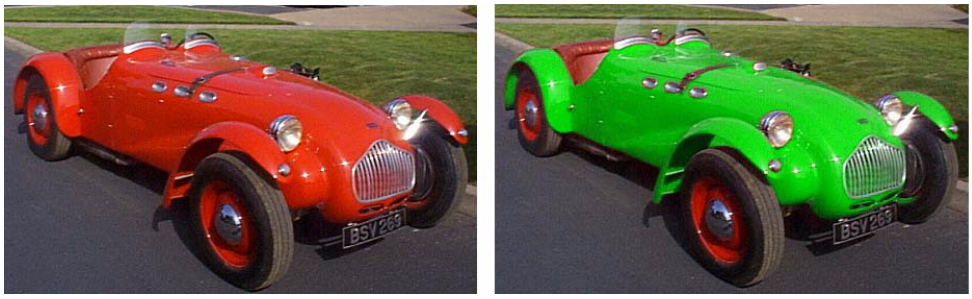
`Color` - specifies the color using its hexadecimal or rgb value. The best results will appear by creating or loading a selection first.

NOTE: *The extreme colors of solid black `(0x000000)` and solid white `(0xffffff)` do not appear correctly when used for `colorize()`. It is recommended that, instead, you use 0x101010, and 0xe0e0e0 or less (for black and white, respectively). Also, totally saturated colors (such as pure red) can create unexpected results.*

`layers` - for PSD files, specifies the layers to be colorized. The layer numbers begin at 0 (background) and go up. For more information see load() on page 101.

Example

```
var image = new Media();
image.load(name @ "car.tga");
image.selection(name @ "mskcar.tga");
image.colorize(color @ 0x009900);
image.save(type @ "jpeg");
```



colorFromImage()

The specified color is converted to the colorspace defined by the destination profile from the colorspace defined by the source profile and returned to the caller. By default, the source profile is the profile embedded in the image. The specified rendering intent is used for the conversion. A source profile may be supplied, and is used if the image has no embedded profile or if the `OverrideEmbedded` parameter is specified as true.

Syntax

```
colorFromImage(
       color @ <color in hexadecimal, rgb, or cmyk>
       [ sourceProfile @ <"filename.icc">]
       [ destProfile @ <"filename.icc">]
       [ intent @ <"rendering intent">
       );
```

Parameters

`color` - specifies the to color to convert using its hexadecimal or rgb value.

`sourceProfile` - specifies the profile used for the source colorspace. Use this parameter to define the color specified in the color parameter if the image has no embedded profile or if the `OverrideEmbedded` flag is set to true. Otherwise, the specified color is defined by the embedded profile.

`destProfile` - specifies the profile used for the destination colorspace.

NOTE: *For more information, see the section about specifying ICC profiles in* *`colorCorrect()` on page 59*.

`intent` - the rendering intent to use for the conversion. This is an optional parameter.

Example

```
rgbColor = image.colorFromImage(color @ 0x00aaaa00,
DestProfile @ "rgb");
```

This example converts the color (red) to the colorspace defined by the profile embedded in image. If image has no embedded profile, an exception is thrown. Assuming the image is a cmyk image with an embedded profile, the resulting color will be the rgb color corresponding to color.

NOTE: *If a Color Profile is associated with an RGB image, this is considered unnecessary data and by default the attached profiles will not be saved to RGB images.*

For more information, please see Appendix B, "MediaRich Color Management.".

colorToImage()

The specified color is converted from the colorspace defined by the source profile to the colorspace defined by the destination profile and returned to the caller. By default, the destination profile is the profile embedded in the image. The specified rendering intent is used for the conversion. A destination profile may be supplied, and will be used if the image has no embedded profile or if the OverrideEmbedded parameter is specified as true.

Syntax

```
colorToImage(
       color @ <color in hexadecimal, rgb, or cmyk>
       [ destProfile @ <"filename.icc">]
       [ sourceProfile @ <"filename.icc">]
       [ intent @ <"rendering intent">
       );
```

Parameters

`color` - specifies the to color to convert using its hexadecimal or rgb value.

`sourceProfile` - specifies the profile used for the source colorspace.

`destProfile` - specifies the profile used for the destination colorspace if the image has no embedded profile or if the overrideembedded flag is set to true.

NOTE: *For more information, see the section about specifying ICC profiles in* *`colorCorrect()` on page 59*.

`intent` - the rendering intent to use for the conversion. This is an optional parameter.

Example

```
cmykcolor = image.colorToImage(color @ 0xaa0000,
SourceProfile @ "rgb");
```

This example converts the color (red) to the colorspace defined by the profile embedded in image. If image has no embedded profile, an exception is thrown. Assuming the image is a CMYK image with an embedded profile, the resulting color will be the CMYK color corresponding to color.

For more information, please see Appendix B, "MediaRich Color Management.".

composite()

Composites the specified foreground (source) image onto the current (background) image. The image specified by source must be loaded separately. The background and source images can be any bit-depth. Transparency is available only for 16-bit, 32-bit, 40-bit (CMYK-A) images with the alpha channel of the source image being used to determine transparency levels.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129. This function also frequently uses* `Media object` *components such as* `getHeight()`, `getWidth()` *and others.*

Syntax

```
composite(
        [ Source @ <user-defined Media object name>]
        [ Name @ <"filename", "virtualfilesystem:/filename">]
        [ Onto @ <true, false>]
        [ Opacity @ <value 0..255>]
        [ Unlock @ <color in hexadecimal or rgb>]
        [ Color @ <color in hexadecimal or rgb>]
        [ Index @ <value 0..16777215>]
        [ Saturation @ <value 0..255>]
        [ FixAlpha @ <true, false>]
        [ PreserveAlpha @ <true, false>]
        [ IgnoreAlpha @ <true, false>]
        [ X @ <pixel>]
        [ Y @ <pixel>]
        [ HandleX @ <"left", "center", "right">]
        [ HandleY @ <"top", "middle", "bottom">]
        [ Tile @ <true, false>]
        [ Blend @ <"blend-type">]
        );
```

NOTE: *Before you can composite an image, you must* `load()` *it.*

Parameters

`Source` - specifies the image using its user-defined Media object name. This parameter does not require quotes.

`Name` - specifies the image by its name and extension (such as "airplane.jpg"). Use this parameter if you are compositing with an image that you have not yet loaded.

If `Source` or `Name` is not specified, MediaRich will perform a color-fill when you also specify the `Color` parameter. For example, if you composite without naming a source, and specify the color green (0x009900), the green will appear composited over the entire background or onto the area of the background as specified through a selection (as with the following example).

```
var image = new Media()
var image2 = new Media();
image.load(name @ "car.tga");
image2.load(name @ "mskcar.tga");
image.selection(source @ image2);
image.composite(color @ 0x009900);
image.save(type @ "jpeg");
```

`Onto` - when specified, the system composites the source onto the loaded image. This way the current media acts like the source, and the loaded one acts like the background. The user can construct a source image and then composite it onto another image without having to cache the source.

NOTE: *Trying to composite an RGB image onto a CMYK image or vice versa results in the process stopping at the `composite()` line with an error.*

`Opacity` - specifies opacity of the source image. The default value is 255 (completely solid).

NOTE: *If the source image already has an alpha channel that renders it less than solid, specifying opacity can only make it less opaque; it cannot override the alpha channel to make it more opaque.*

Specifying a color value for `Unlock` causes the selected foreground (source) image to display only where the specified color value appears in the current (background) image.

Color - colorizes the source image. Any transparency or masking still behaves normally. This allows a source image to be used as a pattern that can be composited in any color, without having to create a new image first. For more information about colorizing an image, see `colorize()`.

If a color palette exists for the source image, you can use the `Index` parameter to colorize the image (as an alternative to the Color parameter).

NOTE: *You cannot specify values for both the `Color` and `Index` parameters.*

`Saturation` - specifies the value used for weighting for the change in saturation for destination pixels. A value of 255 changes the saturation of pixels to the specified color. A value of 128 changes the saturation of a pixel to a mid-value between the pixel's current color and the specified color.

NOTE: *The `Saturation` parameter only functions when the `Blend` parameter is set to "colorize."*

`FixAlpha` - if set to "true, this is equivalent to applying the `fixAlpha()` command. It may be required with some images to get the expected results.

`PreserveAlpha` - when set to true, preserves the alpha channel of the target image as the alpha channel of the resulting image. The default is false.

`IgnoreAlpha` - when set to true, the source is composited onto the target and alpha channel information is ignored. The default is false.

`X` and `Y` - specify the position of the source image, with the center as anchor point. For example, if "`x @ 100, y @ 50`" is specified, the center of the source image will be located at pixel (100,50) on the target image. If these parameters are not specified, the center of the source image is located at pixel (0,0).

`HandleX` and `HandleY` - specify the attachment point of the source image. The default values are center and middle.

`Tile` - when set to true, the source image wraps continuously along both the x and y axis so that it spans the entire target image. The tiling starts in the location specified by the `X` and `Y` and `HandleX` and `HandleY` parameters. If not specified, tiling starts from the target image's center.

NOTE: *If the source image is larger than the target image, setting the `Tile` parameter to true has no effect, unless the source image is sufficiently offset from the center to allow this effect to display.*

`Blend` - specifies the type of blending used to combine the drawn object with the images. Blend modes are: "Normal", "Darken", "Lighten", "Hue", "Saturation", "Color", "Luminosity", "Multiply", "Screen", "Dissolve", "Overlay," "HardLight", "SoftLight", "Difference", "Exclusion", "Dodge", "ColorBurn", "Under", "Colorize" (causes only the hue component of the source to be stamped down on the image.), and "Prenormal".

This function supports CMYK for the following blend modes: "Normal", "Darken", "Lighten", "Screen", "Multiply", "Dissolve", "Overlay", "HardLight", "SoftLight", "Difference", "Exclusion", "Burn", "Dodge", "Under", "Copy", and "PreNormal". The other modes (Hue, Saturation, Color, Luminosity, and Colorize) are not supported for CMYK. You must first convert to RGB using `colorCorrect()` and then perform the composite. Additionally, composite cannot be performed unless both images are either CMYK or RGB.

Example

```
var Target = new Media();
var Source = new Media();
Target.load(name @ "pasta.tga");
Source.load(name @ "logo.tga");
Target.composite(source @ Source, x @ 100, y @ 150);
Target.save(type @ "jpeg");
```

convert()

Converts the image to the specified type/bit-depth. The 8-bit type is not supported, since this involves a much more complex transformation (palette selection, etc.) — instead, use `reduce()`.

When converting images with no alpha channel, the generated alpha channel is based on the background color of the original if the background is set to transparent. Otherwise, the resulting alpha channel is solid white. You can also use the `setColor()` function (placed before the `convert()` function in the MediaScript) to set the background color, with `Transparency` set to *true*.)

NOTE: *`convert()` will convert between CMYK and CMYKA. To convert CMYK color-space to RGB color-space and vice versa use `colorCorrect()`.*

Syntax

```
convert(
        RType @ <"bit-depth">
        [ Dither @ <value 0..10>]
        [ PreserveBackground @ <true, false>]
        [ layers @ <"layer list">] // (PSD files only)
        );
```

Parameters

`Rtype` - specifies the target bit depth. Supported bit-depths are: “Gray-8”, “RGB-15”, “RGB-16”, “RGBA-16”, “RGB-18”, “RGB-24”, “RGBA-32”, “HSV-24”, “HLS-24”, “CMYK-32”, and “CMYKA-40”. The 16-bit type is 5-6-5, while the 16a-bit is 1-5-5-5 with the top bit as an alpha channel.

In addition, the following shortcuts will have default values when used as input parameters:

- Gray -> Gray-8
- RGB -> RGB-24
- RGBA -> RGBA-32
- HSV -> HSV-24
- HLS -> HLS-24
- CMYK -> CMYK-32
- CMYKA -> CMYKA-40

NOTE: *Deprecated parameters include: “Grayscale”, “15-bit”, “16-bit”, “16a-bit”, “24-bit”, “32-bit”.*

`Dither` - determines the level of dithering to use for remapping image pixels to a lower bit-depth.

`PreserveBackground` - when dithering is used, eliminates any pixels in the source image that match the background color from the dithering process in the destination image. This can be used to eliminate fuzzy edges for an object against a solid color background.

`layers` - for PSD files, specifies the layers to be converted. Specify the layers to collapse and the order in which to collapse them. The layer numbers begin at 0 (background) and go up. For more information see "load()" on page 101.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.convert(rtype @ "Grayscale", dither @ 5);
image.save(type @ "jpeg");
```

convolve()

Convolves the image with the specified filter.

Syntax

```
convolve(
        filter @ <"filter list">
        );
```

Parameters

`filter` - specifies the standard filter to be applied. Available filters are:

- "Blur" - standard blur filter.
- "Smooth" - standard smooth filter.
- "Sharpen" - standard sharpen filter.
- "Emboss1" - standard emboss filter.
- "Emboss2" - alternate emboss filter.
- "Edges" - edge filter.

crop()

Crops/resizes the media to a specified size. This function fully supports CMYK.

The background color may vary with this function, depending on the original Media object. If the object has a set background color, or it is set with the setColor() function, MediaRich uses the set color. However, if the object has no set background color, MediaRich does the following:

- For objects with 256 colors or less, MediaRich uses the first color index
- For objects with 15-bit or greater resolution (including the CMYK color-space), MediaRich uses black

Syntax

```
crop(
        [ Xs @ <pixels>, <percentage + "%">]
        [ Ys @ <pixels>, <percentage + "%">]
        [ Xo @ <left pixel>]
        [ Yo @ <top pixel>]
        [ layers @ <"layer list">] // (PSD files only)
        [ PadColor @ <color in hexadecimal or rgb>]
        [ PadIndex @ <value 0..16777215>]
        [ Transparency @ <value 0..255>]
        [ Alg @ <"Normal", "BackColor", "Color", "Alpha">]
        );
```

Parameters

`Xs` and `Ys` - specify the size of the resulting image. The size can be specified either as an absolute, or as a percentage of the original size (percentages must be designated by adding the "%" as in the Syntax example). Where xs or ys is not specified, the original size is used.

`Xo` and `Yo` - specify the position of the top left of the marquee to use. Where either of these is not specified, the marquee is centered on the image.

`layers` - for PSD files, specifies the layers to be cropped.The layer numbers begin at 0 (background) and go up. For more information see load() on page 101.

`Padcolor` or `Padindex` - specifies the color to be used where the new image dimensions extend beyond the current image. If a pad color is not specified, the image's background color is used by default. For more information about setting an image's background color, see setColor() on page 131.

`Transparency` - specifies the transparency (255 is opaque and 0 is transparent) of the padded area's alpha channel. This parameter is useful when the cropped image is used in a `composite()` function.

NOTE: *If the cropped image is not 32-bit before cropping, the transparency information is not used on the next `composite()` function.*

`Alg` - when set to anything other than `"Normal"`, the area specified (or the whole image if no area was defined) is scanned, and the area to crop shrunk accordingly:

- `"BackColor"` trims away the background areas only.
- `"Color"` trims away areas that match the pad color.
- `"Alpha"` trims away areas with transparent alpha channels.

NOTE: *Using `Alg @ Alpha` on an image with no alpha channel, but which has transparency on, will give the same results as `Alg @ BackColor`.*

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.crop(xs @ 20, ys @ 16 + "%", padcolor @ 0xe0e0e0);
image.save(type @ "jpeg");
```

digimarcDetect()

Returns true if the object includes embedded Digimarc information. Otherwise returns false.

Syntax

```
digimarcDetect();
```

Parameters

There are no parameters for this function.

Example

```
var Image = new Media();
Image.load(name @ "peppers.jpg");
if (Image.digimarcDetect() == true)
{
       // Do something if a watermark is dected
       Image.drawText(font @ "Arial", style @ "Bold", text @ "A
DigiMarc watermark has been detected!", size @ 20);
       Image.save(type @ "jpeg");
}
```

digimarcEmbed()

Embeds Digimarc information in the specified image.

Syntax

```
digimarcEmbed(
       [ Type @ <"type">]
       [ CreatorID @ <id number>]
       [ CreatorPin @ <pin number>]
       [ DistributorID @ <id number>]
       [ DistributorPin @ <pin number>]
       [ ImageID @ <id number>]
       [ TransactionID @ <id number>]
       [ Year1 @ <yyyy>]
       [ Year2 @ <yyyy>]
       [ Adult @ <true, false>]
       [ Restricted @ <true, false>]
       [ CopyProtected @ <true, false>]
       [ Durability @ <value 1..16>]
       );
```

Parameters

The following table describes the parameters taken by this method.

Parameter	Description
`Type`	This indicates the type of digimarc you wish to embed. The default is “basic”. Other options are “image”, “transaction”, and “copyright”. The type determines which set of additional parameters that are valid for the watermark.
`CreatorID`	A number which uniquely identifies the creator of the image. The creator ID maps to a profile of the creator, at the Digimarc MarcCentre website. Valid for the following types: basic, image, transaction, and copyright.
`CreatorPin`	This is a unique Personal Identification Number (PIN) issued with the associated CreatorID. This value is used by DWM to check the validity of the CreatorID. Valid for the following types: basic, image, transaction, and copyright.
`DistributerID`	Identifies the organization that distributes the image. This is a numeric value and can be DistributorID may be set to zero to indicate that no distributor ID is to be placed in the watermark. Note that if a DistributorID of zero is specified, then the CreatorID above should not be zero. Valid for the following types: image, transaction, and copyright.
`DistributorPin`	This is a unique Personal Identification Number (PIN) issued with the associated Distributor ID. This value is used by DWM to check the validity of the Distributor ID. Valid for the following types: image, transaction, and copyright.
`ImageID`	This a 24-bit number that uniquely identifies the image (similar to an image catalog number). If no image ID is desired, set the image ID to zero. Valid for the following type: image.
`TransactionID`	This is a 24-bit number that uniquely identifies an instance of the image. For example, if an image was licensed to a publication for a one-time use, the Transaction ID can be used to track that specific licensing transaction vs. the same image licensed to a different customer for 1 year of use. If no Transaction ID is desired, set the Transaction ID to zero.Valid for the following type: transaction.
`Year1, Year 2`	These are the copyright years that are embedded in the image. One or both of these fields can be empty (zero). Valid for the following type: copyright.
`Adult`	Indicates that image contains adult content when set to “TRUE”. The default value is false (empty). Valid for all types.
`Restricted`	Indicates that image has restricted use when set to “TRUE”. The default value is false (empty). Valid for all types.
`CopyProtected`	Indicates that image should not be copied when set to “TRUE”. The default value is false (empty). Valid for all types.
`Durability`	Indicates the “amount of energy” of the watermark, between 1 and 16. The higher the durability value, the more robust the watermark. The default value is 8. Valid for all types.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.digimarcEmbed(Type @ "transaction", CreatorID @ 404407,
CreatorPin @ 32, DistributorID @ 2591, DistributorPin @ 1355,
TransactionID @ 667, Adult @ true, Restricted @ true,
CopyProtected @ true, Durability @ 16);
image.save(type @ "jpeg");
```

discard()

Removes the designated Media object from memory. This function fully supports CMYK image operations.

Syntax

```
discard()
```

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.discard();
```

drawText()

Composites the specified text string onto the image. This function fully supports CMYK image operations.

The foreground color may vary with this function, depending on the original Media object. If the object has a set foreground color, or it is set with the `setColor()` function, MediaRich uses the set color. However, if the object has no set *foreground* color, MediaRich does the following:

- For objects with 256 colors or less, MediaRich uses the last color index
- For objects with 15-bit or greater resolution (including the CMYK color-space), MediaRich uses white

NOTE: *Using `drawText()` within a JavaScript `for` loop can result in initially poor anti-aliasing. To maintain optimal anti-aliasing, place the text object outside the loop.*

Syntax

```
drawText(
       [ Font @ <"font family", "virtualfilesystem:/font family">]
       [ Style @ <"modifier">]
       [ Text @ <"string">]
       [ Color @ <color in hexadecimal or rgb>]
       [ Index @ <value 0..16777215>]
       [ Unlock @ <color in hexadecimal or rgb>]
       [ Saturation @ <value 0..255>]
       [ Size @ <value>]
       [ Justify @ <"left", "center", "right", "justified">]
       [ Wrap @ <pixel-width>]
```

```
[ Opacity @ <value 0..255>]
[ X @ <pixel>]
[ Y @ <pixel>]
[ HandleX @ <"left", "center", "right">]
[ HandleY @ <"top", "middle", "bottom">]
[ Angle @ <angle>]
[ Smooth @ <true, false>]
[ SmoothFactor <0 .. 4>]
[ BaseLine @ <true, false>]
[ spacing @ <+ or ->]
[ Kern @ <true, false>]
[ Line @ <value 0.1 to 10>]
[ Blend @ <"blend-type">]
[ Tile @ <true, false>]
[ Append @ <true, false>]
[ ClearType @ <true, false>] //(windows only)
[ Dpi @ <0.0...10000.0>]
);
```

Parameters

`Font` - specifies the TrueType or PostScript font family name to be used, for example, "Arial". MediaRich supports Type 1 (.pfa and .pfb) PostScript fonts only.

NOTE: *The size of the font in pixels is dependent on the resolution of the resulting image. If the resolution of the image is not set (zero), the function uses a default value of 72 DPI.*

The default location for fonts specified in a MediaScript is the `fonts` file system. Which includes both the `MediaRich Shared\Originals\Fonts` folder and the default system fonts folder. If a MediaScript specifies an unavailable font, MediaRich generates an error.

NOTE: *You can modify the MediaRich server's `local.properties` file to change the default fonts directory. See the MediaRich Administrator's Guide for more information.*

`Style` - specifies the font style. You can use any combination of modifiers. Each modifier should be separated by a space character.

NOTE: *The `Style` parameter is not available if MediaRich is running on Mac, Linux, or Solaris.*

Weight modifiers modify the weight (thickness) of the font. Valid weight values, in order of increasing thickness, are:

- "thin"
- "extralight" or "ultralight"
- "light"
- "normal" or "regular"
- "medium"
- "semibold" or "demibold" ("semi" or "demi" are also acceptable)
- "bold"

- "extrabold" or "ultrabold" ("extra" or "ultra" are also acceptable)
- "heavy" or "black"

Other `Style` parameter values are "Underline", "Italic" or "Italics", and "Strikethru" or "Strikeout").

NOTE: *You can combine `Style` parameter values. For example: `Style @ "Bold Italic"`*

`Text` - specifies the text to be drawn. The text string must be enclosed in quotes. To indicate a line break, insert `\n` into the text.

`Color` - specifies the color to be used for the text. The default value for text color is the image's foreground color. For more information about setting an image's foreground color, see `setColor()`.

If a color palette is available for coloring the text, you can use the `Index` parameter to colorize the image (as an alternative to the Color parameter).

NOTE: *You cannot specify values for both the `Color` and `Index` parameters.*

`Unlock` - specifies a color value that determines which pixels are displayed in the overlaid source image. Using this parameter causes the selected foreground (source) image to display only where the specified color value appears in the current (background) image.

`Saturation` - specifies the value used for the weighting for the change in saturation for destination pixels. A value of 255 changes the saturation of pixels to the specified color. A value of 128 changes the saturation of a pixel to a mid-value between the pixel's current color and the specified color.

NOTE: *The `Saturation` parameter only functions when the `Blend` parameter is set to "colorize."*

`Size` - sets the point size of the font to be used. The default size is 12.

`Justify` - specifies how the text will be justified. The default is "center". Other options are "left", "right", and "justified". (The "justified" option is available for Windows only.) This parameter only affects text with multiple lines.

`Wrap` - specifies a value used to force a new line whenever the text gets longer than the specified number of pixels (in this case correct word breaking is used).

`Opacity` - specifies opacity of the text. The default value is 255 (completely solid).

`X` and `Y` - specify the text's position on the image, based on text's anchor point. The default value is the center point of the image.

`Handlex` and `Handley` - specify the anchor point of the text (for example, Handlex = left/center/right, Handley= top/middle/bottom) relative to the placement point of the image (as specified by the `X` and `Y` parameters described above). The default values are center and middle.

`Angle` - allows the text to be rotated clockwise by the specified angle (in degrees).

`Line` - specifies line spacing. The default spacing between lines of text is 1.5.

`Smooth` - specifies that the text is drawn with five-level anti-aliasing.

`SmoothFactor` - specifies the power of two for image scale-based smoothing. If "1" is specified, the text will be drawn at twice the specified size and scaled down. If "2" is specified, the text is drawn at four times the size. This scaling produces smoother text for renderers with poor anti-aliasing at smaller text sizes. The `Smooth` parameter must be set to "true" for this parameter to have any effect.

`Baseline` - if specified, the text is treated as though it is always the height of the largest character. This allows text to be aligned between different calls to the function. The distance, in pixels, between the baselines of two lines of text is 1.5 times the point-size of the text. Thus for 30-point text the line spacing is 45 pixels. If this parameter is not specified, this function measures the actual height of the text and centers it accordingly.

`Spacing` - adjusts the spacing between the text characters. The default is 0. A negative value draws the text characters closer together.

`Kern` - if set to "true", which is the default, it optimizes the spacing between text characters. If you do not want to use kerning, specify this parameter as "false".

NOTE: *PostScript fonts store the kerning information in a separate file with a .afm extension. This file must be present in order for kerning to be applied to the text.*

`Blend` - specifies the type of blending used to combine the drawn object with the images. Blend options are: "Normal", "Darken", "Lighten", "Hue", "Saturation", "Color", "Luminosity", "Multiply", "Screen", "Dissolve", "Overlay", "HardLight", "SoftLight", "Difference", "Exclusion", "Dodge", "ColorBurn", "Under", "Colorize" (causes only the hue component of the source to be stamped down on the image), and "Prenormal".

`Tile` - if set to "true", the text wraps continuously along both the x and y axis so that it spans the entire target image. The tiling starts in the location specified by the `X` and `Y` and `HandleX` and `HandleY` parameters. If not specified, tiling starts from the target image's center.

NOTE: *If the source image is larger than the target image, setting the `Tile` parameter to true has no effect, unless the source image is sufficiently offset from the center to allow this effect to display.*

`Append` - if set to "true", `drawText()` appends the text of the previous call to the specified text string.

NOTE: *Append works best when drawing a single line of left-justified text, as subsequent calls to `drawText()` will not maintain the wrap or justification information.*

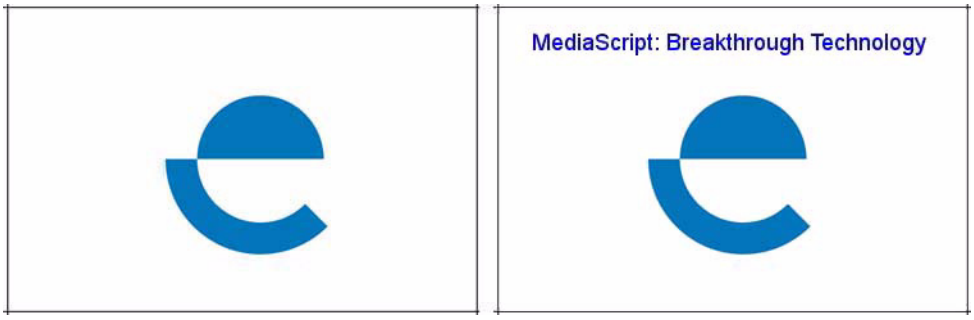
`ClearType` - if specified as "true", the Windows ClearType text renderer will be used if available.

`DPI` - specifies the DPI used for text rendering. The default value is 72.

NOTE: *The `DPI` parameter is not available if MediaRich is running on Mac, Linux, or Solaris.*

Example

```
var image = new Media();
image.load(name @ "logobg.tga");
image.drawText(Font @ "Arial", Style @ "Bold", Text @ "MediaScript:
Breakthrough Technology", Size @ 18, Color @ 0x0000FF, x @ 185, y @
30, Smooth @ true, Kern @ true);
image.save(type @ "jpeg");
```



dropShadow()

Adds a drop shadow to the image based on its alpha channel. The effects are best seen when compositing the results onto another image. This function fully supports CMYK image operations.

NOTE: *Occasional unexpected results can often be corrected with the fixAlpha() command. See "fixAlpha()" on page 80 for more details.*

Syntax

```
dropShadow(
      [ ResizeCanvas @ <true, false>]
      [ layers @ <"layer list">] // (PSD files only)
      [ Opacity @ <value 0..255>]
      [ Blur @ <value 0..30>]
      [ Dx @ <number of pixels>]
      [ Dy @ <number of pixels>]
      [ Color @ <color in hexadecimal or rgb>]
      [ Index @ <value 0..16777215>]
      );
```

Parameters

`ResizeCanvas` - provides for the canvas of the image to be automatically enlarged to encompass the shadow produced. The image's background color will be used for the additional area. For more information about setting an image's background color, see `setColor()`.

NOTE: *The `Enlarge` parameter has been deprecated.*

`layers` - for PSD files, specifies the layers to be affected. The layer numbers begin at 0 (background) and go up. For more information see "load()" on page 101.

`Opacity` - defines the level of transparency for the shadow. The default opacity is 255, which is completely solid. The shadow affects the alpha channel of the image as well as the visible channels.

`Blur` - adds blurring that results in a shadow with a more diffused look. Note, however, that the larger the blur value, the more processing is required.

`Dx` and `Dy` - specify the offset of the shadow from the original, where positive values shift the shadow down and to the right.

`Color` - specifies the color to be used for the shadow. The default is the foreground color.

`Index` - colorizes the image shadow using an available color palette for the source image (as an alternative to the `Color` parameter).

NOTE: *You cannot specify values for both the `Color` and `Index` parameters.*

Example

```
var image = new Media();
var image2 = new Media();
image.load(name @ "peppers.tga");
image2.drawText(font @ "Arial", style @ "Bold", text @ "Fresh
Peppers!", angle @ 30, color @ 0x00ccff, size @ 36, smooth @ true,
baseline @ true, kern @ true);
image2.dropShadow(opacity @ 255, blur @ 2, dx @ 5, dy @ 15, color @
0x000000);
image.composite(source @ image2);
image.save(type @ "jpeg");
```

ellipse()

Draws and positions an ellipse on the image based on the specified parameters. This method accepts all `composite()` parameters except `HandleX` and `HandleY`.

The foreground color may vary with this function, depending on the original Media object. If the object has a set foreground color, or it is set with the `setColor()` function, MediaRich uses the set color. However, if the object has no set *foreground* color, MediaRich does the following:

- For objects with 256 colors or less, MediaRich uses the last color index
- For objects with 15-bit or greater resolution, MediaRich uses white

NOTE: *Using `ellipse()` to mask frames within a JavaScript `for` loop can result in initially poor anti-aliasing. To maintain optimal anti-aliasing, place the masking ellipse outside the loop.*

Syntax

```
ellipse(
        X @ <pixel>
        Y @ <pixel>
        Rx @ <value>
        Ry @ <value>
        [ Opacity @ <value 0..255>]
        [ Unlock @ <color in hexadecimal or rgb>]
        [ Color @ <color in hexadecimal or rgb>]
        [ Index @ <value 0..16777215>]
        [ Saturation @ <value 0..255>]
        [ PreserveAlpha @ <true, false>]
        [ Blend @ <"blend-type">]
        [ Width @ <value>]
        [ Smooth @ <true, false>]
        [ Fill @ <true, false>]
        );
```

Parameters

`X` - specifies (in pixels) the *x* axis coordinate for the center point of the ellipse. This parameter is required and has no default value.

`Y` - specifies (in pixels) the *y* axis coordinate for the center point of the ellipse. This parameter is required and has no default value.

`Rx` - specifies (in pixels) the radius of the ellipse on the *x* axis. This parameter is required and has no default value.

`Ry` - specifies (in pixels) the radius of the ellipse on the *y* axis. This parameter is required and has no default value.

`Opacity` - specifies opacity of the drawn object. The default value is 255 (completely solid).

`Unlock` - if set to true, causes the ellipse to display only where the specified color value appears in the current (background) image. The default is false.

`Color` - specifies the color to be used for the ellipse. The default is the foreground color.

`Index` - colorizes the ellipse using the available color palette for the source image (as an alternative to the `Color` parameter).

NOTE: *You cannot specify values for both the `Color` and `Index` parameters.*

`Saturation` - specifies a value used for weighting for the change in saturation for destination pixels. A value of 255 changes the saturation of pixels to the specified color. A value of 128 changes the saturation of a pixel to a mid-value between the pixel's current color and the specified color.

NOTE: *The `Saturation` parameter only functions when the `Blend` parameter is set to "colorize."*

`PreserveAlpha` - if set to true, preserves the alpha channel of the target image as the alpha channel of the resulting image. The default is false.

`Blend` - specifies the type of blending used to combine the drawn object with the images. Blend options are: "Normal", "Darken", "Lighten", "Hue", "Saturation", "Color", "Luminosity", "Multiply", "Screen", "Dissolve", "Overlay," "HardLight", "SoftLight", "Difference", "Exclusion", "Dodge", "ColorBurn", "Under", "Colorize" (causes only the hue component of the source to be stamped down on the image), and "Prenormal".

NOTE: *"Burn" has been deprecated. "ColorBurn" results in the same blend.*

`Width` - specifies the thickness (in pixels) of the line that describes the ellipse. The default is 1.

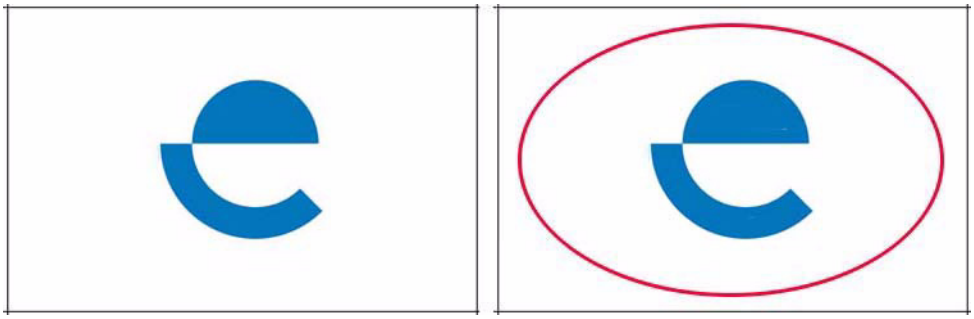
NOTE: *If the `Fill` parameter is set to true, `Width` is ignored.*

`Smooth` - if set to true, makes the edges of the ellipse smooth, preventing a pixellated effect. The default is false.

`Fill` - fills in the ellipse with the color specified by the `Color` or `Index` parameter. The default is false.

Example

```
var image = new Media();
image.load(name @ "logobg.tga");
image.ellipse(X @ 272, Y @ 180, Rx @ 50, Ry @ 30, Opacity @ 128,
Color @ 0x66CCFF, Saturation @ 128, Blend @ "Hue", Smooth @ true,
Fill @ false);
image.save(type @ "jpeg");
```



embeddedProfile()

Returns `true` if the media has an embedded ICC profile, `false` if not.

Syntax

```
<object name>.embeddedProfile();
```

Parameters

This function has no parameters.

Example

```
if (image.embeddedProfile() == false)
image.colorCorrect(destProfile @ "sRGB.icc");
{ ...
```

equalize()

Equalizes the relevant components of the media. Equalization takes the used range of a component and expands it to fill the available range. This can be applied to both indexed and non-indexed images.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see* `selection()`.

Syntax

```
equalize(
        [Brightness @ <-1.00 to 20.00>]
        [Saturation @ <-1.00 to 20.00>]
        );
```

Parameters

`Brightness` and `Saturation` - specify values that are given in terms of clip-value. Clip-value is the percentage of pixels that can lie outside the measured range before expansion and whose value is therefore clipped in the process. The valid values are 0 to 20 and –1, although values between 0.5 and 1.0 generally produce the most favorable results.

NOTE: *As a special case, specifying a clip-value of –1 applies histogram equalization to that channel. Histogram equalization is a much harsher method, but effectively maximizes the amount of visible information in an image.*

Example

```
var image = new Media();
image.load(name @ "car.tga");
image.equalize(brightness @ 10, saturation @ 5);
image.save(type @ "jpeg");
```

exportChannel()

Exports a single channel of the source as a grayscale image. This function fully supports the CMYK color-space.

NOTE: *This function was formerly named* `exportGun()`*, which has been deprecated.*

Syntax

```
exportChannel(
        Channel @ <"channelname">
        [ layers @ <"layer list">] // (PSD files only)
        );
```

Parameters

Valid channel names are:

- “Blue”, “Green”, “Red”, “Alpha”,
- “Cyan”, “Magenta”, “Yellow”, “Black” (CMYK-space)
- “Brightness”, “Saturation”, “Hue” (HSV-space)
- “Brightness2”, “Saturation2”, “Hue2” (HLS-space)

The default value is Blue.

`layers` - for PSD files, specifies the layers to be exported. The layer numbers begin at 0 (background) and go up. For more information see “load()” on page 101.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.exportChannel(channel @ "green");
image.save(type @ "jpeg");
```

NOTE: *Comparing the original CMYK image and the newly generated images in Photoshop will show the exact inverse results of what Photoshop displays for the separate channels.*

fixAlpha()

This function has been removed from MediaScript. If it is encountered, no operation will be performed.

Adjusts the RGB components of an image relative to its alpha channel. This should be done when an alpha channel has been manually created for an image. This command will frequently correct unexpected results in functions that utilize the alpha channel.

Syntax

```
fixAlpha();
```

Example

```
var image = new Media();
image.load(name @ "pasta.tga");
image.fixAlpha();
image.save(type @ "jpeg");
```

flip()

Flips the media vertically or horizontally. This function fully supports images within the CMYK color space.

Syntax

```
flip(
        Axis @ <"Horizontal, Vertical">
        [ layers @ <"layer list">] // (PSD files only)
        );
```

Parameters

`Axis` - designates along which axis (*horizontal* or *vertical*) to flip the media.

`layers` - for PSD files, specifies the layers to be affected. The layer numbers begin at 0 (background) and go up. For more information see "load()" on page 101.

Example

```
var image = new Media();
image.load(name @ "pasta.tga");
image.flip(axis @ "horizontal");
image.save(type @ "jpeg");
```

frameAdd()

Adds the given frame(s) to the specified Media object. If the Media object already contains one or more images, any frames added are cropped and converted to match the first frame in the media. This function fully supports the CMYK color space.

Before you can add a frame, you must `load()` the image you want to add.

NOTE: *When using `frameAdd()` within a JavaScript `for` loop: including `Draw` functions (`line()`, `ellipse()`, etc.) within the loop to mask other frames can result in initially poor anti-aliasing.*

Syntax

```
frameAdd(
        [ Source @ <user-defined Media object name>]
        [ Name @ <"filename", "virtualfilesystem:/filename">]
        [ Duration <1 .. 300000>]
        );
```

Parameters

`Source` - specifies the image to add by its user-defined Media object name.If you are adding an image that you have not yet loaded, use the `Name` parameter to refer to that image by its name and extension (such as "airplane.jpg").

`Name` - if you or the administrator has set up virtual file systems, you can use this parameter to add frames from that file system. Virtual file systems are defined in the MediaRich server's `local.properties` file. See the *MediaRich Administrator's Guide* for more information.

For example, if you define "MyImages:" to represent the path "C:/Images/MyImages/" in the local.properties file, you can use files from the MyImages directory with the `frameAdd()` function:

```
image.frameAdd(name @ "MyImages:/split.tga");
```

NOTE: *The `Name` parameter has been deprecated.*

`Duration` - specifies the frame duration in seconds. This sets the duration of the frame in our internal structure. Specifying this does not currently do anything useful.

Example

```
var image = new Media();
var image2 = new Media();
image.load(name @ "peppers.tga");
image2.load(name @ "Bears.tga");
image.frameAdd(Source @ image2);
image.reduce();
image.save(type @ "gif");
```

getBitsPerSample()

Returns the number of bits per sample.

Syntax

```
<object name>.getBitsPerSample();
```

Parameters

This function has no parameters.

Example

```
if (image.getBitsPerSample() == 8)
{...
```

getBytesPerPixel()

Returns the number of bytes per pixel.

Syntax

```
<object name>.getBytesPerPixel();
```

Parameters

This function has no parameters.

Example

```
if (image.getBytesPerPixel() == 3)
{...
```

NOTE: *If you want MediaRich to return the bit-depth of the image, use the* `getImageFormat()` *function.*

getFrame()

Returns a Media object for the specified frame (if available), otherwise returns "undefined".

Syntax

```
<object name>.getFrame(
        <frame offset>
);
```

Parameters

This function takes the specified frame offset (starting from 1) as an argument.

Example

```
var image = new Media();
image.load(name @ "Images/clock.gif");   // Load an animated GIF with
four frames
image2 = image.getFrame(2);
image2.save(name @ "frame2.gif");
```

getFrameCount()

Returns the number of frames in an animation.

Syntax

```
<object name>.getFrameCount();
```

Parameters

This function has no parameters.

Example

```
for (x = 0;x < image.getFrameCount();x++)
{...
```

getHeight()

Returns the vertical size in pixels.

Syntax

```
<object name>.getHeight();
```

Parameters

This function has no parameters.

Example

```
if (image.getHeight() == 480)
{...
```

getImageFormat()

Returns a string representing the image type: “8 Bit Palette”, “8 Bit Grayscale”, “15 Bit 5-5-5”, “16 Bit 5-6-5”, “16 Bit 1-5-5-5”, “18 Bit 6-6-6”, “24 Bit”, “32 Bit RGBA”, “32 Bit HSV”, “32 Bit HLS”, or “Undefined”.

NOTE: *`GetImageFormat()` has been deprecated. `getPixelFormat()` is the preferred function and will be supported in future versions of MediaScript.*

Syntax

```
<object name>.getImageFormat();
```

Parameters

This function has no parameters.

Example

```
var image = new Media();
image.load(name @ "peppers.psd");
if(image.getImageFormat() == "24 Bit")
{...
```

getInfo()

Returns the system version information. It is recommended for advanced users only.

Syntax

```
<object name>.getInfo(
       <"argument">
       );
```

Parameters

It takes one of these arguments as a string: “all”, “devices”, “commands”, or “filing”.

Example

```
var image = new Media();
error(image.getInfo("all"));
```

getLayer()

Returns a Media object for the specified layer (if available); otherwise returns “undefined”. It takes the specified layer index (starting from zero) as an argument.

Syntax

```
<object name>.getLayer(
       <layer number>
       );
```

Parameters

`layer number` - specifies the desired layer of the Media object.

Example

```
var image = new Media();
var newimage = new Media();
newimage = image.getLayer(2);
```

getLayerBlend()

Returns the blending mode of the media layer with the specified layer index (if available).

Syntax

```
<object name>.getLayerBlend(
       <layer index>
       );
```

Parameters

`layer index` - specifies the desired layer index (starting from 0).

The blend modes are: “Normal”, “Darken”, “Lighten”, “Hue”, “Saturation”, “Color”, “Luminosity”, “Multiply”, “Screen”, “Dissolve”, “Overlay,” “HardLight”, “SoftLight”, “Difference”, “Exclusion”, “Dodge”, “ColorBurn”, “Under”, and “Colorize”.

NOTE: *“Burn” has been deprecated. “ColorBurn” results in the same blend.*

Example

```
var image = new Media();
image.load(name @ "peppers.psd");
if(image.getLayerBlend(0) == "Saturation")
{...
```

getLayerCount()

Returns the total number of layers for the media.

Syntax

```
<object name>.getLayerCount();
```

Parameters

This function has no parameters.

Example

```
for(x = 0;x < image.getLayerCount();x++)
{...
Layer = image.getLayer(x);
...}
```

getLayerEnabled()

Returns `true` if the named layer is enabled (visible), `false` if not. If you use the `collapse()` function without naming specific layers, MediaRich collapses all enabled layers and ignores disabled layers. Use the `getLayerEnabled()` function to determine if a layer is enabled or not. Use the `setLayerEnabled()` function or the eye icon in Photoshop to enable/disable a layer.

Syntax

```
<object name>.getLayerEnabled(
        <layer index>
);
```

Parameters

`layer index` - specifies the desired layer index (starting from 0).

Example

```
if (image.getLayerEnabled(2) == false)
image.setLayerEnabled(2, true);
...}
```

getLayerHandleX()

Returns the HandleX value ("left", "center", or "right") of the media layer with the specified index (if available). HandleX refers to the selected layer's attachment point on the *x*-axis.

Syntax

```
<object name>.getLayerHandleX(
        <layer index>
        );
```

Parameters

`layer index` - specifies the desired layer index (starting from 0).

Example

```
var image = new Media();
image.load(name @ "peppers.psd");
if(image.getLayerHandleX(0) == "Center")
{...
```

getLayerHandleY()

Returns the HandleY value ("top", "middle", or "bottom") of the media layer with the specified index (if available). HandleY refers to the selected layer's attachment point on the *y*-axis.

Syntax

```
<object name>.getLayerHandleY(
        <layer index>
        );
```

Parameters

`layer index` - specifies the desired layer index (starting from 0).

Example

```
var image = new Media();
image.load(name @ "peppers.psd");
if(image.getLayerHandleY(0) == "Middle")
{...
```

getLayerIndex()

Returns the index of the media layer with the specified layer name (if available).

Syntax

```
<object name>.getLayerIndex(
        <"layer name">
        );
```

Parameters

The only parameter specifies the desired layer name.

Example

```
var image = new Media();
image.load(name @ "peppers.psd");
if(image.getLayerIndex("GreenPepper") == 2)
{...
```

getLayerName()

Returns a string with the name of the media layer (if available). It takes the specified layer index (starting from zero) as an argument.

Syntax

```
<object name>.getLayerName(
        <layer or "layername">
        );
```

Parameters

`layer index` - specifies the desired layer index (starting from 0).

Example

```
var image = new Media();
image.load(name @ "peppers.psd");
if(image.getLayerName(2) == "GreenPepper")
{...
```

getLayerOpacity()

Returns the opacity of the media layer with the specified index (if available). For more information about opacity settings, see "composite()" on page 63.

Syntax

```
<object name>.getLayerOpacity(
       <layer index>
       );
```

Parameters

`layer index` - specifies the desired layer index (starting from 0).

Example

```
var image = new Media();
image.load(name @ "peppers.psd");
if(image.getLayerOpacity(2) == 50)
{...
```

getLayerX()

Returns the X offset of the media layer with the specified index (if available).

NOTE: *X and Y layer offsets determine relative positions of layers to each, and are used by the* `collapse()` *function.*

Syntax

```
<object name>.getLayerX(
       <layer index>
       );
```

Parameters

`layer index` - specifies the desired layer index (starting from 0).

Example

```
var image = new Media();
image.load(name @ "peppers.psd");
if(image.getLayerX(2) == 25)
{...
```

getLayerY()

Returns the Y offset of the media layer with the specified index (if available).

NOTE: *X and Y layer offsets determine relative positions of layers to each, and are used by the* `collapse()` *function.*

Syntax

```
<object name>.getLayerY(
        <layer index>
        );
```

Parameters

`layer index` - specifies the desired layer index (starting from 0).

Example

```
var image = new Media();
image.load(name @ "peppers.psd");
if(image.getLayerY(2) == 25)
{...
```

getMetaData()

Returns a metadata string of the specified format associated with a Media object. MediaRich allows users to assign arbitrary key/value pairs to any Media object using the `setMetaData()` function. The `getMetaData()` function returns the value as a string.

Syntax

```
var xmlDoc = <object name>.getMetadata(<"format">);
```

Parameters

The specified format is the key of the key/value pair. Valid values are “Exif”, “IPTC”, or “XMP”.

Example

```
var image = new Media();
image.load(name @ "peppers.psd");
image.getMetaData("Exif");
{...
```

getPalette()

Returns an array of integers containing the colors in the palette or null if the image does not have a palette. The RGB components of these colors can be obtained using the RGBColor object defined in `sys/color.ms`. If the image has no palette the array is empty.

Syntax

```
palColors = <object name>.getPalette();
```

Parameters

This function has no parameters.

Example

```
#include "sys:color.ms"
.
.
.
var colors = media.getPalette();
if (colors.length >= 1)
{
        var rgb = new RGBColor(colors[ 0] );
        print("red is " + rgb.red);
}
```

getPaletteSize()

Returns the number of colors in the palette or 0 if the image does not have a palette.

Syntax

```
var nColors = <object name>.getPaletteSize();
```

Parameters

This function has no parameters.

getPixel()

Returns the color value, omitting any alpha channel. For RGB images, this will be a 24-bit color value. For CMYK, a 32-bit color value.

Syntax

```
<object name>.getPixel(
        X @ <pixel>
        Y @ <pixel>
        [ layers @ <"layer list">] // (PSD files only)
        );
```

Parameters

`X` and `Y` - specify the coordinates of the target pixel. The top-left corner of an image is represented by the coordinates 0,0.

`layers` - for PSD files, specifies the layers to be included.The layer numbers begin at 0 (background) and go up. For more information see load() on page 101.

Example

```
var image = new Media();
image.load(name @ "peppers.psd");
if(image.getPixel(X @ 25, Y @ 100)== rgb(255,0,0))
{...
```

getPixelFormat()

Returns a string representing the image type: “Gray-8”, “RGB-15”, “RGB-16”, “RGBA-16”, “RGB-18”, “RGB-24”, “RGBA-32”, “HSV-24”, “HLS-24”, “CMYK-32”, “CMYKA-40”.

Syntax

```
<object name>.getPixelFormat();
```

Parameters

This function has no parameters.

Example

```
var image = new Media();
image.load(name @ "peppers.psd");
if(image.getPixelFormat() == "RGB-24")
{...
```

getPixelTransparency()

Returns the value for the alpha channel or 255 if there is no alpha channel. This function fully supports the media object within the CMYK color space.

Syntax

```
<object name.>.getPixelTransparency(
        X @ <pixel>
        Y @ <pixel>
        [layers @ <"layer list">] // (PSD files only)
);
```

Parameters

`X` and `Y` - specify the coordinates of the target pixel. The top-left corner of an image is represented by the coordinates 0,0.

`layers` - for PSD files, specifies the layers to be included. The layer numbers begin at 0 (background) and go up. For more information see “load()” on page 101.

getPopularColor()

Returns the 24-bit color value (0 - 16,777,215) of the color that appears most frequently in the named object for the RGB color-space.

Syntax

```
<object name>.getPopularColor(
        [Precise @ <true, false>]
        [layers @ <"layer list">] // (PSD files only)
        );
```

Parameters

`Precise` - if set to “false”, which is the default, the color returned will be a close approximation of the actual color that appears most often in the image.

`layers` - for PSD files, specifies the layers to be included. The layer numbers begin at 0 (background) and go up. For more information see "load()" on page 101.

Example

```
var image = new Media();
image.load(name @ "peppers.psd");
if(image.getPopularColor()== rgb(255,0,0))
{...
```

getResHorizontal()

Returns the horizontal resolution in DPI.

Syntax

```
<object name>.getResHorizontal();
```

Parameters

This function has no parameters.

Example

```
if (image.getResHorizontal() == 72)
{...
```

getResVertical()

Returns the vertical resolution in DPI.

Syntax

```
<object name>.getResVertical();
```

Parameters

This function has no parameters.

Example

```
if (image.getResVertical() == 72)
{...
```

getSamplesPerPixel()

Returns the number of samples per pixel.

Syntax

```
<object name>.getSamplesPerPixel();
```

Parameters

This function has no parameters.

Example

```
if (image.getSamplesPerPixel() == 3)
{...
```

getWidth()

Returns the horizontal size in pixels.

Syntax

```
<object name>.getWidth();
```

Parameters

This function has no parameters.

Example

```
if (image.getWidth() == 480)
{...
```

getXmlInfo()

Returns an xml document contain the installed file formats. This document looks like following:

```
<fileformats>
      <fileformat>
            <name>format name</name>
            <version>format version</version>
            <extensions>comma separated list of
extensions</extensions>\n";
            <modes>read,write</modes>
      </fileformat>
...
</fileformats>
```

Syntax

```
xmlString = media.getXmlInfo();
```

Parameters

This function has no parameters.

glow()

Produces a glow or halo around the image. It is similar to the `dropShadow()` method and is based on the alpha channel of the image. Its effects are best seen when compositing the results onto another image.

NOTE: *Occasional unexpected results can often be corrected with the `fixAlpha()` command. See `fixAlpha()` for more details.*

The foreground and background colors may vary with this function, depending on the original Media object. If the object has foreground and background colors, or such colors are set with the `setColor()` function, MediaRich uses the set colors including the CMYK color-space.

However, if the object has no set *background* color, MediaRich does the following:

- For objects with 256 colors or less, MediaRich uses the first color index
- For objects with 15-bit or greater resolution (including the CMYK color-space), MediaRich uses black

If the object has no set *foreground* color, MediaRich does the following:

- For objects with 256 colors or less, MediaRich uses the last color index
- For objects with 15-bit or greater resolution (including the CMYK color-space), MediaRich uses white

Syntax

```
glow(
        Blur @ <value 0..30>
        [ Size @ <value 1..30>]
        [ Halo @ <value 0..30>]
        [ Opacity @ <value 0..255>]
        [ Color @ <color in hexadecimal or rgb>]
        [ Index @ <value 0..16777215>]
        [ ResizeCanvas @ <true, false>]
        [ layers @ <"layer list">] // (PSD files only)
        );
```

Parameters

`Blur` - adds a specified blur to the shadow that gives it a more diffused look. Note, however, that the larger the blur value, the more processing is required.

`Size` - specifies how large (in pixels) the glow surrounding the image should be.

`Halo` - specifies the gap between the image and the start of the glow. The value for halo must always be smaller than the size of the glow. The default value is zero.

`Opacity` - defines the level of transparency for the shadow. The default opacity is 255, which is completely solid.

NOTE: *The shadow affects the alpha channel of the image as well as the visible channels.*

Color - defines the color of the glow, and the default is the foreground color.

`Index` - colorizes the glow using an index value from the available color palette for the source image (as an alternative to the `Color` parameter).

NOTE: *You cannot specify values for both the `Color` and `Index` parameters.*

`ResizeCanvas` - automatically resizes the canvas of the image to encompass the shadow produced. The image's background color will be used for the additional area. For more information about setting an image's background color, see "setColor()" on page 131.

`layers` - for PSD files, specifies the layers to be affected. The layer numbers begin at 0 (background) and go up. For more information see "load()" on page 101.

Example

```
var image = new Media();
var image2 = new Media();
image.load(name @ "peppers.tga");
image2.drawText(font @ "Arial", style @ "Bold", text @ "Fresh
Peppers!", angle @ 0, color @ 0x00ccff, size @ 36, smooth @ true,
baseline @ false, kern @ true);
image2.glow(Blur @ 4, Size @ 8, Halo @ 0, Opacity @ 220, Color @
0xFFFF00, ResizeCanvas @ false);
image.composite(source @ image2);
image.save(type @ "jpeg");
```

gradient()

Composites a color gradient onto the source image. This method accepts all `composite()` parameters except `HandleX` and `HandleY`.

Syntax

```
gradient(
       [adjust @ <true, false>],
       [style=<Linear,Radial,Angle,Reflected,Diamond>]
       [angle=<0..360>]
       [scale=<10..150>]
       [reverse]
       [color1=<RGBcolor>]
       [color2=<RGBcolor>]
       [gradient=<RedGreen,
                  VioletOrange,
                  BlueRedYellow,
                  BlueYellowBlue,
                  OrangeYellowOrange,
                  VioletGreenOrange,
                  YellowVioletOrangeBlue,
                  Copper,
                  Chrome,
                  Spectrum>]
```

```
[ opacity=<value 0..255>]
[ blend=<blend-type>]
[ layers @ <"layer list">] // (PSD files only)
```

Parameters

`adjust` - when specified, all the other parameters except `color1`, `color2`, `gradient`, and `reverse` (which have their usual meaning) are ignored. The image is then interpreted as a grayscale image which is then passed through the specified gradient, giving a new false-color image. This operates the same way as the GradientMap adjustment layer in Photoshop.

`style` - specifies a common style for the gradient. The available styles are: Linear, Radial, Angle, Reflected, and Diamond.

`angle` - specifies the value of the angle at which the gradient is applied. This value can range from 0 to 360, to indicate the degree of the angle.

`scale` - specifies a scale to be applied to the gradient. This value can range from 10 to 150.

`reverse` - when specified, reverses the direction of the applied gradient.

`color1` and `color2` - used as an alternative to specifying a common gradient, these parameters specify a custom gradient created by blending the two specified RGB colors. If only one of these parameters is specified, a gradient is created that blends between the specified color and transparent.

`gradient` - specifies a common color gradient that blend two or three standard colors. The available gradients are RedGreen, VioletOrange, BlueRedYellow, BlueYellowBlue, OrangeYellowOrange, VioletGreenOrange, YellowVioletOrangeBlue, Copper, Chrome, Spectrum.

NOTE: *If color1 and/or color2 are specified together with gradient, a parameter clash error occurs.*

`opacity` - specifies opacity of the source image. The default value is 255 (completely solid).

NOTE: *If the source image already has an alpha channel that renders it less than solid, specifying opacity can only make it less opaque; it cannot override the alpha channel to make it more opaque.*

`blend` - specifies the type of blending used to combine the drawn object with the images. Blend modes are: “Normal”, “Darken”, “Lighten”, “Hue”, “Saturation”, “Color”, “Luminosity”, “Multiply”, “Screen”, “Dissolve”, “Overlay,” “HardLight”, “SoftLight”, “Difference”, “Exclusion”, “Dodge”, “ColorBurn”, “Under”, “Colorize” (causes only the hue component of the source to be stamped down on the image), and “Prenormal”.

This function supports CMYK for the following blend modes: “Normal”, “Darken”, “Lighten”, “Screen”, “Multiply”, “Dissolve”, “Overlay”, “HardLight”, “SoftLight”, “Difference”, “Exclusion”, “Burn”, “Dodge”, “Under”, “Copy”, and “PreNormal”. The other modes (Hue, Saturation, Color, Luminosity, and Colorize) are not supported for CMYK. You must first convert to RGB using `colorCorrect()` and then perform the composite. Additionally, composite cannot be performed unless both images are either CMYK or RGB.

`layers` - for PSD files, specifies the layers to be affected. The layer numbers begin at 0 (background) and go up. For more information see load() on page 101.

importChannel()

Imports the specified source image (treated as a grayscale) and replaces the selected channel in the original. It is important that both images must be the same size. Before you can import an image, you must `load()` it.

NOTE: *This function was formerly named `importGun()`, which has been deprecated.*

NOTE: *Color value parameters to functions supporting CMYK are interpreted as CMYK colors if the raster to which they are applied is CMYK.*

Syntax

```
importChannel(
       Channel @ <"channel name">
       [ Source @ <user-defined Media object name>]
       [ layers @ <"layer list">] // (PSD files only)
       [ RType @ <"bit-depth">]
       );
```

Parameters

`Source` - specifies the image to add by its user-defined Media object name.

`layers` - for PSD files, specifies the layers to be included. The layer numbers begin at 0 (background) and go up. For more information see "load()" on page 101.

`Rtype` - specifies the target bit depth. The supported bit depths: "RGB-24", "RGBA-32", "CMYK-32", "CMYKA-40", "Gray-8", "RGB-15", "RGB-16", "RGBA-16", "RGB-18", "Alpha-8", "HLS-24", "HSV-24".

Valid channel names are:

- "Blue", "Green", "Red", "Alpha",
- "Cyan", "Magenta", "Yellow", "Black" (CMYK-space)
- "Brightness", "Saturation", "Hue" (HSV-space)
- "Brightness2", "Saturation2", "Hue2" (HLS-space)

The default value is Blue.

NOTE: *If you attempt to import an alpha channel into a 24-bit image, it will automatically be converted to a 32-bit image.*

Example

```
var image1 = new Media();
var image2 = new Media();
image1.load(name @ "peppers.tga");
image2.load(name @ "Bears.tga");
image2.scale(xs @ image1.getWidth(), ys @ image1.getHeight());
image1.importChannel(channel @ "red", source @ image2);
image1.save(type @ "jpeg");
```

infoText()

Returns the information about text.

Syntax

```
infoText(
        [font @ <"font">],
        [size @ <"size">],
        [style @ <"style">])
```

Return Values

ascent - the font ascent.

descent - the font descent

height - the font height

averageWidth - the average character width

maxWidth - the maximum character width

weight - the font weight

italic - returns “1” if italic

underlined - returns “1” if underlined

strikeout - returns “1” if strikeout

overhang - extra width that may be added to some fonts by GDI

Parameters

`font` - specifies the TrueType or PostScript font family name to be used, for example, “Arial”. For more information about how MediaRich methods work with fonts and font files, see the “drawText()” on page 71.

`size` - sets the point size of the font to be used. The default size is 12.

`style` - specifies the font style. You can use any combination of modifiers. Each modifier should be separated by a space character.

NOTE: *The `Style` parameter is not available if MediaRich is running on Mac, Linux, or Solaris.*

Weight modifiers modify the weight (thickness) of the font. Valid weight values, in order of increasing thickness, are:

- “thin”
- “extralight” or “ultralight”
- “light”
- “normal” or “regular”
- “medium”
- “semibold” or “demibold” (“semi” or “demi” are also acceptable)
- “bold”
- “extrabold” or “ultrabold” (“extra” or “ultra” are also acceptable)
- “heavy” or “black”

Other `Style` parameter values are “Underline”, “Italic” or “Italics”, and “Strikethru” or “Strikeout”).

NOTE: *You can combine `Style` parameter values. For example: `Style @ "Bold Italic"`*

line()

Draws a line across the image based on the specified parameters. This method accepts all composite() parameters except `HandleX` and `HandleY`.

The foreground color may vary with this function, depending on the original Media object. If the object has a set foreground color, or it is set with the `setColor()` function, MediaRich uses the set color. However, if the object has no set *foreground* color, MediaRich does the following:

- For objects with 256 colors or less, MediaRich uses the last color index
- For objects with 15-bit or greater resolution, MediaRich uses white

NOTE: *Using `line()` to mask frames within a JavaScript `for` loop can result in initially poor anti-aliasing. To maintain optimal anti-aliasing, place the masking line outside the loop.*

Syntax

```
line(
        X1 @ <pixel>,
        Y1 @ <pixel>,
        X2 @ <pixel>,
        Y2 @ <pixel>,
        [ Opacity @ <value 0..255>]
        [ Unlock @ <color in hexadecimal or rgb>]
        [ Color @ <color in hexadecimal or rgb>]
        [ Index @ <value 0..16777215>]
        [ Saturation @ <value 0..255>]
        [ PreserveAlpha @ <true, false>]
        [ Blend @ <"blend-type">]
        [ Width @ <value>]
        [ Smooth @ <true, false>]
        );
```

Parameters

`X1` - indicates (in pixels) the *x* axis coordinate of the line start point. This parameter is required and has no default value.

`Y1` - indicates (in pixels) the *y* axis coordinate of the line start point. This parameter is required and has no default value.

`X2` - indicates (in pixels) the *x* axis coordinate of the line end point. This parameter is required and has no default value.

`Y2` - indicates (in pixels) the *y* axis coordinate of the line end point. This parameter is required and has no default value.

`Opacity` - specifies opacity of the drawn object. The default value is 255 (completely solid).

`Unlock` - if set to "true", causes the line to display only where the specified color value appears in the current (background) image. The default is false.

`Color` - specifies the color of the line.

`Index` - colorizes the line using the available color palette from the source image (as an alternative to the `Color` parameter).

NOTE: *You cannot specify values for both the `Color` and `Index` parameters.*

`Saturation` - specifies a value used for weighting for the change in saturation for destination pixels. A value of 255 changes the saturation of pixels to the specified color. A value of 128 changes the saturation of a pixel to a mid-value between the pixel's current color and the specified color.

NOTE: *The `Saturation` parameter only functions when the `Blend` parameter is set to "colorize."*

`PreserveAlpha` - when set to "true", preserves the alpha channel of the target image as the alpha channel of the resulting image. The default is false.

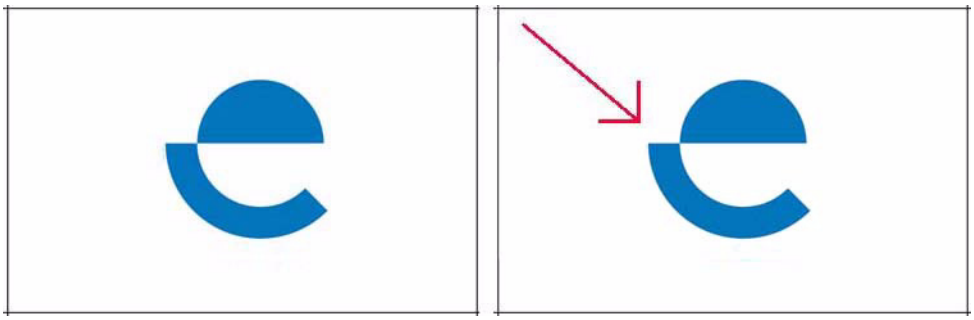
`Blend` - specifies the type of blending used to combine the drawn object with the images. Blend options are: "Normal", "Darken", "Lighten", "Hue", "Saturation", "Color", "Luminosity", "Multiply", "Screen", "Dissolve", "Overlay," "HardLight", "SoftLight", "Difference", "Exclusion", "Dodge", "ColorBurn", "Under", "Colorize" (causes only the hue component of the source to be stamped down on the image), and "Prenormal".

`Width` - specifies the thickness (in pixels) of the line. The default is 1.

`Smooth` - when set to "true", makes the edges of the line smooth, preventing a pixellated effect. The default is false.

Example

```
var image = new Media();
image.load(name @ "logobg.tga");
image.line(X1 @ 45, Y1 @ 15, X2 @ 135, Y2 @ 90, Width @ 3);
image.line(X1 @ 135, Y1 @ 60, X2 @ 135, Y2 @ 90, Width @ 3);
image.line(X1 @ 105, Y1 @ 90, X2 @ 135, Y2 @ 90, Width @ 3);
image.save(type @ "jpeg");
```



load()

Loads an image into the Media object from the specified file. The following file formats support the CMYK color-space: ".psd", ".tif", and ".jpg".

NOTE: *In MediaRich version 3.6, `load()` no longer does any color conversion. For instance, additional parameters for Color Profile Specifications `srcProfile`, `destProfile`, and `intent` parameters are no longer supported. You must now explicitly call `convert()` or `colorConvert()` to change the type of an image.*

Syntax

```
load(
      [name @ <"filename", "http://server_name/../filename",
"ftp://username:password@ftp.server_name/../ filename",
"ftp://ftp.server_name/../filename", "virtualfilesystem:/filename">]
      [type @ <"typename">]
      [detect @ <true, false>]
      [transform @ <true, false>] // (FPX files only)
      [layers @ <"layer list">] // (PSD files only)
      [temporaryFileName @ <"filename">]
      [collapsed @ <true, false>] // (PSD files only)
      [VisibleOnly @ <true, false>] // (PSD files only)
      [PreviewAlpha @ <true, false>] // (PSD files only)
      [transform @ <true, false>] // (FPX files only)
      [fillalpha @ <true, false>] // (PNG files only)
      [screengamma @ <value 0..10>] // (PNG files only)
      [waplook @ <true, false>] // (WBMP files only)
      [dpi @ <value 1..32767>] // (EPS, PDF, and PS files only)
      [LoadMetadata @ <true, false>]
      );
```

Parameters

`name` - specifies the filename and path (full or relative) of the file to be loaded. By default, MediaRich looks for media in the `read` file system which points to the following directory: `MediaRich/Shared/Originals/Media`.

You can modify the MediaRich server's `local.properties` file to change the default Media directory. See the *MediaRich Administrator's Guide* for more information. You can also load a file from an HTTP or FTP URL using the `name` parameter.

NOTE: *The functionality of loading files from HTTP or FTP sources is disabled by default for security reasons. Contact your MediaRich Administrator to enable this functionality.*

`type` - specifies the expected file type; otherwise the type is derived from the file extension. Valid type names are: "bmp", "eps", "flashpix", "gif", "jpeg", "png", "pict", "pcx", "pdf", "photoshop", "ps", "tiff", "targa", and "wbmp".

NOTE: *Some image formats are module and/or platform specific. Please visit the support section of the Equilibrium website for the most current list.*

`detect` - indicates that if a matching file type is not found, or if the load returns with a FileMangled or FileTypeWrong error, the system will attempt to automatically determine the file's type and load it accordingly.

`LoadMetadata` - if specified as "true", any Exif, IPTC, or XMP Metadata associated with the image is loaded. The default is "false". For more information about MediaRich's metadata support, see Appendix A, "MediaRich Metadata Support."

Default Page Range and Specification of Page Range for PS, EPS, and PDF

`dpi` - use this parameter for EPS, PDF, or PS source files to determine the size of the image after it is loaded. The default is 72. In addition, when loading one of these source files only the first page/frame of the file is loaded by default. To load more than just the first page, the "frames" parameter must be specified with a page range. To load the entire file when the length is not known, either of the following will work:

- frames "1-0"
- frames "1-999999" (assuming there aren't more than 999999 pages)

Frames can also select some subset of all pages in the file, and a single frame may also be specified. In addition, "frame" is a valid alias for "frames". Any of the following are valid:

- frames "3-9"
- frames 7
- frame 12
- frame "4-9"

Intermediate File Specification

Instead of rendering pages into the source Media, you can use the `load()` function to render eps/pdf/ps/ai files into a multi-frame TIFF file. This method avoids having to read an entire multi-page file into memory at once.

`temporaryFileName` - specifies the TIFF file to render into, and specifies that file not be deleted after the operation completes. The normal `load()` operation is not otherwise affected, so pages may still be rendered into the source Media as well. If the user wishes to not render any pages into the source Media, they can specify that only frame 0 (zero) be rendered. The following is an example of how to render into a TIFF file without adding any frames to the Media:

```
img.load(name @ "anImage.ps", temporaryFileName @ "foo.tif",
frame @ 0);
```

NOTE: *You will get an error message if you try to load a Photoshop image that includes an adjustment layer.*

Additional Parameters for Photoshop Files

When loading a Photoshop file, MediaRich by default loads a single raster, created by the Photoshop application. Photoshop creates this raster based on the visible layers contained in the PSD file.

`collapsed` - when set to "false", overrides the single raster default loading.

`layers` - loads the specified layers. Specify the layers to collapse and the order in which to collapse them. The layer numbers begin at 0 (background) and go up. To specify all layers (including non-visible layers) use the wild-card notation "*". The `visibleOnly()` function may used to load only the visible layers of a PSD file.

NOTE: *MediaRich loads only the image data from the layers and ignores all other effects. To preserve such effects, merge the effects into the layer data in Photoshop. You can specify the layers out of order, and they are composited accordingly.*

The layer list must be contained in quotes and consists of comma-separated entries. You can specify ranges ("0-2") or individual layers ("0,2").

NOTE: *When you specify a comma-separated list of layers, do not leave any spaces after the commas.*

If the Photoshop file has named layers, you can use the layer names (up to 31 characters) in place of layer numbers. You can also use the "*" as a wildcard when specifying layers. For example:

```
image.load(name @ "horizontal.psd", layers @ "B*")
```

This line of script loads all layers whose names begin with "B" (such as Boy, Baseball, Ballcap, and so on). The layers command is case-sensitive, so the example line of script will not flip layers that begin with a lowercase "b".

NOTE: *All Photoshop Adjustment Layers must be merged into the layer image data prior to use in MediaRich.*

`VisibleOnly` - when set to "true" and loading photoshop layers (`collapsed` is set to "false"), only the layers designated as visible are loaded. The default is false.

`PreviewAlpha` - when set to "false" and layers are not specified, the loaded image's preview will not contain the preview alpha channel.

Parameters for FPX Files

`transform` - when set to "true", performs any transformations that are embedded in the FPX file. The default is false.

Parameters for PNG Files

`fillalpha` - when set to "true", fills transparent and translucent pixels with the image's background color. The default is false.

`screengamma` - specifies a floating gamma point, causing the reader to perform a gamma correction if the file contains a specified gamma value. The default is 0 (no correction).

Parameter for WBMP Files

`waplook` - when set to "true", sets the image's palette to simulate the look of an LCD screen on an actual WAP device.

Example

```
var image = new Media();
image.load(name @ "peppers.tga", type @ "targa");
image.save(type @ "jpeg");
```

loadAsRgb()

The `loadAsRgb()` function is an add-on to the Media object that acts exactly like `load()` does when an RGB file is read. When a CMYK file is read, the images in the file are converted to RGB. This function is defined in `Sys/media.ms`.

Syntax

```
loadAsRgb(
        [name @ <"filename">]
        [type @ <"typename">]
        [detect @ <true, false>]
        [transform @ <true, false>] // (FPX files only)
        [layers @ <"layer list">] // (PSD files only)
        [fillalpha @ <true, false>] // (PNG files only)
        [screengamma @ <value 0..10>] // (PNG files only)
        [waplook @ <true, false>] // (WBMP files only)
        [dpi @ <value 1..32767>] // (EPS, PDF, and PS files only)
        [sourceProfile @ <"filename.icc">]
        [destProfile @ <"filename.icc">]
        [intent @ <"rendering intent">
        [overrideEmbedded @ <true, false>]
);
```

Parameters

`name`, `type`, `detect`, `transform`, `layers`, `fillalpha`, `screengamma`, `waplook`, and `dpi` - these parameters operate the same as for the `load()` function. For more information, see "load()" on page 101.

`sourceProfile`, `destProfile`, `intent`, and `overrideEmbedded` - used to determine how this conversion is performed. If any of these parameters are not supplied, the current defaults (as specified in the properties file) are used instead.

NOTE: *If the `destProfile` parameter is specified, the resulting image will be in the colorspace of the specified profile. If this profile does not have an RGB colorspace, the resulting image will NOT be an RGB image.*

Example

```
#include "Sys/media.ms"
        var image = new Media();
        image.loadAsRgb(name @ "myCmykImage.tif");
```

makeCanvas()

This function creates a "blank" Media object of the specified dimensions and fully supports the CMYK color-space.

Syntax

```
makeCanvas(
        [ Xs @ <width in pixels>]
        [ Ys @ <height in pixels>]
        [ Rtype @ <bit-depth>]
        [ FillColor @ <color in hexadecimal or rgb>]
        [ Transparency @ <true, false>]
```

Parameters

`Xs` and `Ys` - specify the width and height of the canvas in pixels. If `Xs` or `Ys` is not specified, a 1x1 canvas is created. If only one of `Xs` and `Ys` is specified, the unspecified parameter is assumed to be the same as the specified one (a square canvas is created).

`Rtype` - specifies the bit-depth. Supported bit-depths are: "RGB-24", "RGBA-32", "CMYK-32", "CMYKA-40", "Gray-8", "RGB-15", "RGB-16", "RGBA-16", "RGB-18", "Alpha-8", "HLS-24", and "HSV-24". The default bit-depth is "RGBA-32" (RGB, 32-bit).

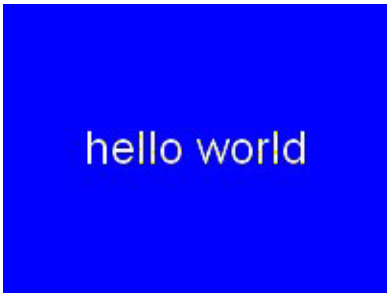
NOTE: *The 16-bit type, is 5-6-5, while the 16a-bit is 1-5-5-5 with the top bit as an alpha channel.*

`FillColor` - determines the color value given to each pixel in the generated canvas. If FillColor is not specified, each pixel is set to black.

`Transparency` - set to "true", the canvas' pixels are all set as transparent and FillColor is used as both the foreground and background color. If Transparency is set to false, the canvas' pixels are set as solid. `FillColor` is used for the foreground color, and the background color is black. Transparency is set to false by default.

Example

```
var image = new Media();
var text = new Media();
image.makeCanvas(Xs @ 200, Ys @ 150, FillColor @ 0x0000ff);
text.makeText(text @ "hello world", font @ "Arial", style @ "Bold",
size @ 24, smooth @ true, color @ 0xffffff);
image.composite(source @ text);
image.save(type @ "jpeg");
```



makeText()

This command, instead of compositing text onto the target image, creates a new image that includes just the text. The image produced is always 32-bit. This function fully supports the CMYK color-space.

NOTE: *Using `makeText()` within a JavaScript for loop can result in initially poor anti-aliasing. To maintain optimal anti-aliasing, place the text object outside the loop.*

Syntax

```
makeText(
        [ Font @ <"font family", "virtualfilesystem:/font family">]
        [ Style @ <"modifier">]
        [ Text @ <"string">]
        [ Color @ <color in hexadecimal or rgb>]
        [ Rtype @ <bit-depth>]
        [ Size @ <value 1..4095>]
        [ Justify @ <"left", "center", "right", "justified">]
        [ Wrap @ <pixel-width>]
        [ Angle @ <angle>]
        [ Smooth @ <true, false>]
        [ SmoothFactor <0 .. 4>]
        [ BaseLine @ <true, false>]
        [ Kern @ <true, false>]
        [ Line @ <value 01. to 10>]
        [ DPI @ <resolution>]
        [ Fillcolor @ <color in hexadecimal or rgb>]
        [ ClearType @ <true, false>] //(windows only)
        [ FitText <true, false>]
        );
```

Parameters

`Font` - specifies the TrueType or PostScript font family to be used, for example, “Arial”. MediaRich supports Type 1 (.pfa and .pfb) PostScript fonts only.

NOTE: *The size of the font in pixels is dependent on the resolution of the resulting image. If the resolution of the image is not set (zero), the function uses a default value of 72 dpi.*

The default location for fonts specified in a MediaScript is the `fonts` file system. Which includes both the MediaRich `Shared\Originals\Fonts` folder and the default system fonts folder. If a MediaScript specifies an unavailable font, MediaRich generates an error.

NOTE: *You can modify the MediaRich server’s `local.properties` file to change the default fonts directory. See the MediaRich Administrator’s Guide for more information.*

`Style` - specifies the font style. You can use any combination of modifiers. Each modifier should be separated by a space character.

NOTE: *The `Style` parameter is not available if MediaRich is running on Mac, Linux, or Solaris.*

Weight modifiers modify the weight (thickness) of the font. Valid weight values, in order of increasing thickness, are:

- "thin"
- "extralight" or "ultralight"
- "light"
- "normal" or "regular"
- "medium"
- "semibold" or "demibold" ("semi" or "demi" are also acceptable)
- "bold"
- "extrabold" or "ultrabold" ("extra" or "ultra" are also acceptable)
- "heavy" or "black"

Other `Style` parameters are "Underline", "Italic" or "Italics", and "Strikethru" or "Strikeout".

NOTE: *You can combine `Style` parameters as necessary. For example: `Style @ "Bold Italic"`*

`Text` - specifies the text to be drawn. The text string must be enclosed in quotes. To indicate a line break, use `\n`.

`Color` - specifies the color to be used for the text. The default value for text color is white. For more information about setting a foreground color, see "setColor()" on page 131.

`Rtype` - specifies the target bit depth. Supported bit-depths are: "Gray-8", "RGB-15", "RGB-16", "RGBA-16", "RGB-18", "RGB-24", "RGBA-32", "HSV-24", "HLS-24", "CMYK-32", "CMYKA-40". The 16-bit type, is 5-6-5, while the 16a-bit is 1-5-5-5 with the top bit as an alpha channel.

In addition, the following shortcuts will have default values when used as input parameters:

- Gray -> Gray-8
- RGB -> RGB-24
- RGBA -> RGBA-32
- HSV -> HSV-24
- HLS -> HLS-24
- CMYK -> CMYK-32
- CMYKA -> CMYKA-40

NOTE: *Deprecated `Rtype` values include: "Grayscale", "15-bit", "16-bit", "16a-bit", "24-bit", "32-bit".*

`Size` - sets the point size of the font to be used, and its default value is 12 and the maximum is 4095.

`Justify` - specifies how the text will be justified. The default is "center". Other options are "left", "right-hand", and "justified". (The "justified" option is available on Windows only.) This parameter only affects text with multiple lines.

`Wrap` - if specified, its value forces a new line whenever the text gets longer than the specified number of pixels (in this case correct word breaking is used).

`Angle` - allows the text to be rotated clockwise by the specified angle (in degrees).

`Line` - specifies the line spacing. The default spacing between lines of text is 1.5.

`Smooth` - specifies that the text is drawn with five-level anti-aliasing.

`SmoothFactor` - specifies the power of two for image scale-based smoothing. If "1" is specified, the text will be drawn at twice the specified size and scaled down. If "2" is specified, the text is drawn at four times the size. This scaling produces smoother text for renderers with poor anti-aliasing at smaller text sizes. The `Smooth` parameter must be set to "true" for this parameter to have any effect.

`Baseline` - if specified, the text is treated as though it is always the height of the largest character. This allows text to be aligned between different calls to the function. The distance, in pixels, between the baselines of two lines of text is 1.5 times the point-size of the text. Thus for 30-point text the line spacing is 45 pixels. If this parameter is *not* specified, `makeText` measures the actual height of the text and centers it accordingly.

`Kern` - if set to "true", optimizes the spacing between text characters. By default this is set to true. If you do not want to use kerning, this must be specified as false.

NOTE: *PostScript fonts store the Kerning information in a separate file with a .afm extension. This file must be present in order for kerning to be applied to the text.*

`DPI` - allows you to specify the image resolution in dots per inch (DPI).

NOTE: *The `DPI` parameter is not available if MediaRich is running on Mac, Linux, or Solaris.*

`Fillcolor` - specifies the color to be used for the background. The default value is black.

`ClearType` - if specified as "true", the Windows ClearType text renderer will be used if available.

`FitText` - if specified as "true", any empty space surrounding the generated text is removed.

Example

```
var image = new Media();
image.makeText(text @ "Your message goes here.",Font @ "Arial",
Style @ "Bold", color @ 0xffff00, smooth @ true);
image.save(type @ "jpeg");
```

measureText()

Returns an array of offsets where each character would be drawn for a single line of text. If more that one line of text is specified (by including "\n") then only the first line of text is measured. This method is available for Windows only.

Syntax

```
measureText(
        [text @ <"string">],
        [font @ <"font family">],
        [size @ <value 1..4095>],
        [style@ <"modifier">],
        [spacing @ <"spacing">],
        [smooth @ <true, false>],
        [ClearType @ <"cleartype">], //Windows only
        [kern @ <true, false>]
);
```

Parameters

`Font` - specifies the TrueType or PostScript font family name to be used, for example, "Arial". MediaRich supports Type 1 (.pfa and .pfb) PostScript fonts only.

NOTE: *The size of the font in pixels is dependent on the resolution of the resulting image. If the resolution of the image is not set (zero), the function uses a default value of 72 dpi.*

The default location for fonts specified in a MediaScript is the `fonts` file system. Which includes both the MediaRich `Shared\Originals\Fonts` folder and the default system fonts folder. If a MediaScript specifies an unavailable font, MediaRich generates an error.

NOTE: *You can modify the MediaRich server's local.properties file to change the default fonts directory. See Installing and Managing MediaRich for more information.*

`Style` - specifies the font style. You can use any combination of modifiers. Each modifier should be separated by a space character.

NOTE: *The `Style` parameter is not available if MediaRich is running on Mac, Linux, or Solaris.*

Weight modifiers modify the weight (thickness) of the font. Valid weight values, in order of increasing thickness, are:

- "thin"
- "extralight" or "ultralight"
- "light"
- "normal" or "regular"
- "medium"
- "semibold" or "demibold" ("semi" or "demi" are also acceptable)
- "bold"
- "extrabold" or "ultrabold" ("extra" or "ultra" are also acceptable)
- "heavy" or "black"

Other `Style` parameters are “Underline”, “Italic” or “Italics”, and “Strikethru” or “Strikeout”).

NOTE: *You can combine `Style` parameters. For example: `Style @ "Bold Italic"`*

`Text` - specifies the text to be drawn. The text string must be enclosed in quotes. To indicate a line break, insert `\n` into the text.

`Size` - sets the point size of the font to be used. The default size is 12.

`Spacing` - adjusts the spacing between the text characters. The default is 0.

NOTE: *A negative `Spacing` value draws the text characters closer together.*

`Smooth` - specifies that the text is drawn with five-level anti-aliasing.

`ClearType` - if specified as “true”, the Windows ClearType text renderer will be used if available.

`Kern` - if set to “true”, optimizes the spacing between text characters. By default this is set to true. If you do not want to use kerning, this must be specified as false.

NOTE: *PostScript fonts store the Kerning information in a separate file with a .afm extension. This file must be present in order for kerning to be applied to the text.*

noiseAddNoise()

Applies random pixels to an image to simulate a noise effect.

NOTE: *This function is “selection aware,” so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see “selection()” on page 129.*

Syntax

```
image.noiseAddNoise(
       [Amount @ <value 1..999>]
       [Gaussian @ <true, false>]
       [Grayscale @ <true, false>]
       );
```

Parameters

`Amount` - indicates the intensity of the effect. The default is 32.

`Gaussian` - toggles the Gaussian distribution effect on or off. The default is false (off).

`Grayscale` - applies the monochromatic scale to the affected pixels. The default is false (normal color).

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.noiseAddNoise(Amount @ 15, Gaussian @ true, Grayscale @ true);
image.save(type @ "jpeg");
```

otherHighPass()

Applies an effect opposite that of blurGaussianBlur() — it replaces each pixel with the difference between the original pixel and a Gaussian-blurred version.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
otherHighPass(
      [Radius @ <value, 10..250>]
      );
```

Parameters

`Radius` - specifies the radius of the Gaussian blur aspect of the effect. The default is 10.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.otherHighPass(Radius @ 50);
image.save(type @ "jpeg");
```

otherMaximum()

Replaces the pixels within the radius with the brightest pixel in that radius, thereby amplifying the lighter areas of the image.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
otherMaximum(
        [ Radius @ <value 1..10>]
        );
```

Parameters

`Radius` - determines the extent of the effect. The default is 1 (minimal effect).

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.otherMaximum(Radius @ 2);
image.save(type @ "jpeg");
```

otherMinimum()

Replaces the pixels within the radius with the darkest pixel in that radius, thereby amplifying the darker areas of the image.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
otherMinimum(
        [ Radius @ <value 1..10>]
        );
```

Parameters

`Radius` - determines the extent of the effect. The default is 1 (minimal effect).

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.otherMinimum(Radius @ 2);
image.save(type @ "jpeg");
```

pixellateFragment()

Makes and offsets four copies of the image.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
pixellateFragment(
        [ Radius @ <value 1..16>]
        );
```

Parameters

`Radius` - determines the extent of the offset, with `1` indicating the minimum offset. The default is 4.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.pixellateFragment(Radius @ 2);
image.save(type @ "jpeg");
```

pixellateMosaic()

Pixellates the image, with pixel size determined by the `Radius` parameter.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
pixellateMosaic(
      [ Size <2..64>]
      );
```

Parameters

`Size` - determines the resulting pixel size. The default is 8.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.pixellateMosaic(Radius @ 10);
image.save(type @ "jpeg");
```

polygon()

Draws and positions a polygon on the image based on the specified parameters. This method accepts all composite() parameters except `HandleX` and `HandleY`.

The foreground color may vary with this function, depending on the original Media object. If the object has a set foreground color, or it is set with the setColor() function, MediaRich uses the set color.

However, if the object has no set *foreground* color, MediaRich does the following:

- For objects with 256 colors or less, MediaRich uses the last color index
- For objects with 15-bit or greater resolution, MediaRich uses white

NOTE: *Using `polygon()` to mask frames within a JavaScript `for` loop can result in initially poor anti-aliasing. To maintain optimal anti-aliasing, place the masking polygon outside the loop.*

Syntax

```
polygon(
        Points @ <"x,y;x,y;x,y;x,y">,
        [Opacity @ <value 0..255>]
        [Unlock @ <color in hexadecimal or rgb>]
        [Color @ <color in hexadecimal or rgb>]
        [Index @ <value 0..16777215>]
        [Saturation @ <value 0..255>]
        [PreserveAlpha @ <true, false>]
        [Blend @ <"type">]
        [Width @ <value>]
        [Smooth @ <true, false>]
        [Fill @ <true, false>]
```

Parameters

`Points` - describes each point of the polygon, using absolute coordinate points. Each pair of coordinates is separated from the next by a semicolon. This parameter is required and has no defaults.

NOTE: *To create a closed polygon, the first set of coordinates and the last set of coordinates must be identical. For example, the parameter* `Points @ "16,20;180,160;120,229;16,20"` *describes a closed triangle.*

`Opacity` - specifies opacity of the drawn object. The default value is 255 (completely solid).

`Unlock` - if set to "true", causes the polygon to display only where the specified color value appears in the current (background) image. The default is false.

`Color` - sets the color of the polygon. If a color palette exists for the source image, you can use the `Index` parameter to set the color of the polygon (as an alternative to the `Color` parameter).

NOTE: *You cannot specify values for both the* `Color` *and* `Index` *fields.*

`Saturation` - specifies the value used for weighting for the change in saturation for destination pixels. A value of 255 changes the saturation of pixels to the specified color. A value of 128 changes the saturation of a pixel to a mid-value between the pixel's current color and the specified color.

NOTE: *The* `Saturation` *parameter only functions when the* `Blend` *parameter is set to "colorize."*

`PreserveAlpha` - if set to "true", preserves the alpha channel of the target image as the alpha channel of the resulting image. The default is false.

`Blend` - specifies the type of blending used to combine the drawn object with the images. Blend options are: "Normal", "Darken", "Lighten", "Hue", "Saturation", "Color", "Luminosity", "Multiply", "Screen", "Dissolve", "Overlay", "HardLight", "SoftLight", "Difference", "Exclusion", "Dodge", "ColorBurn", "Under", "Colorize" (causes only the hue component of the source to be stamped down on the image), and "Prenormal".

NOTE: *"Burn" has been deprecated. "ColorBurn" results in the same blend.*

`Width` - specifies the thickness (in pixels) of the line that describes the polygon. The default is 1.

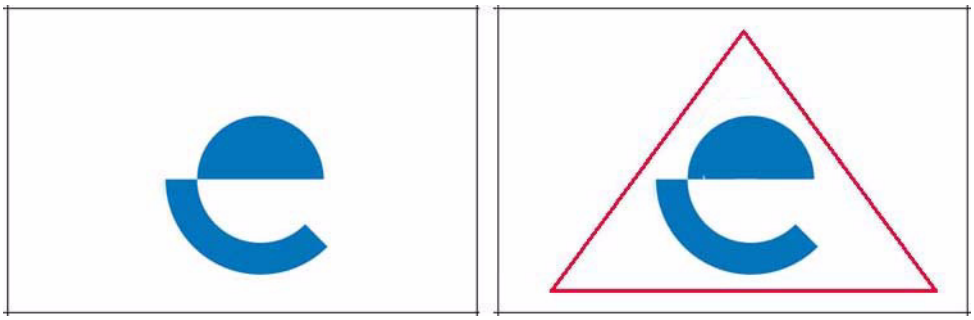
NOTE: *If the `Fill` parameter is set to true, `Width` is ignored.*

`Smooth` - if set to "true", makes the edges of the polygon smooth, preventing a pixellated effect. The default is false.

`Fill` - if set to "true", fills in the polygon with the color specified by the `Color` or `Index` parameter. The default is false.

Example

```
var image = new Media();
image.load(name @ "logobg.tga");
image.polygon(points @ "200,20;350,222;50,222;200,20", width @ 3);
image.save(type @ "jpeg");
```



quadWarp()

Moves the corners of the source image to the specified locations, warping the image accordingly. The top left corner of the source image is represented by the coordinates 0,0.

NOTE: *This is a linear transformation, so while it can be used to "fake" small 3D rotations, for greater angles, the lack of perspective will become apparent.*

This function fully supports the CMYK color-space.

Syntax

```
quadWarp(
        [ Smooth @ <true, false>]
        [ TopLeftX @ <position>]
        [ TopLeftY @ <position>]
        [ BotLeftX @ <position>]
        [ BotLeftY @ <position>]
        [ BotRightX @ <position>]
        [ BotRightY @ <position>]
        [ TopRightX @ <position>]
        [ TopRightY @ <position>]
        [ layers @ <"layer list">] // (PSD files only)
        );
```

Parameters

`Smooth` - provides for smooth edges when warping the image using non-right angles.

`TopLeftX` and `TopLeftY` - represent the upper left corner of the area to be warped. The default is the original image's upper-left corner.

`TopRightX` and `TopRightY` - represent the upper right corner of the area to be warped. The default is the original image's upper-right corner.

`BotLeftX` and `BotLeftY` - represent the lower left corner of the area to be warped. The default is the original image's lower-left corner.

`BotRightX` and `BotRightY` - represent the lower right corner of the area to be warped. The default is the original image's lower-right corner.

`layers` - for PSD files, specifies the layers to be affected. The layer numbers begin at 0 (background) and go up. For more information see "load()" on page 101.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.quadWarp(TopLeftX @ -10, TopLeftY @ -20, TopRightX @ 440,
TopRightY @ 480, BotLeftX @ -40, BotLeftY @ 780, BotRightX @ 640,
BotRightY @ 0, smooth @ true);
image.save(type @ "jpeg");
```

rectangle()

Draws and positions a rectangle on the image based on the specified parameters. This method accepts all composite() parameters except `HandleX` and `HandleY`.

The foreground color may vary with this function, depending on the original Media object. If the object has a set foreground color, or it is set with the setColor() function, MediaRich uses the set color.

However, if the object has no set *foreground* color, MediaRich does the following:

- For objects with 256 colors or less, MediaRich uses the last color index
- For objects with 15-bit or greater resolution, MediaRich uses white

NOTE: *Using `rectangle()` to mask frames within a JavaScript `for` loop can result in initially poor anti-aliasing. To maintain optimal anti-aliasing, place the masking rectangle outside the loop.*

Syntax

```
rectangle(
        X @ <pixel>,
        Y @ <pixel>,
        Xs @ <pixel>,
        Ys @ <pixel>,
        [ Opacity @ <value 0..255>]
        [ Unlock @ <color in hexadecimal or rgb>]
        [ Color @ <color in hexadecimal or rgb>]
        [ Index @ <value 0..16777215>]
        [ Saturation @ <value 0..255>]
        [ PreserveAlpha @ <true, false>]
        [ Blend @ <"blend-type">]
        [ Width @ <value>]
        [ Angle @ <value -360..360>]
        [ Smooth @ <true, false>]
        [ Fill @ <true, false>]
        );
```

Parameters

`X` - indicates (in pixels) the *x* axis coordinate of the upper left corner of the rectangle. This parameter is required and has no default value.

`Y` - indicates (in pixels) the *y* axis coordinate of the upper left corner of the rectangle. This parameter is required and has no default value.

`Xs` - indicates (in pixels) the *x* axis coordinate of the lower right corner of the rectangle, relative to the upper left corner. This parameter is required and has no default value.

`Ys` - indicates (in pixels) the *y* axis coordinate of the lower right corner of the rectangle, relative to the upper left corner. This parameter is required and has no default value.

`Opacity` - specifies opacity of the drawn object. The default value is 255 (completely solid).

`Unlock` - if set to "true", causes the rectangle to display only where the specified color value appears in the current (background) image. The default is false.

`Color` - sets the color of the rectangle.

`Index` - colorizes the line using the available color palette from the source image (as an alternative to the `Color` parameter).

NOTE: *You cannot specify values for both the `Color` and `Index` parameters.*

`Saturation` - specifies a value used for weighting for the change in saturation for destination pixels. A value of 255 changes the saturation of pixels to the specified color. A value of 128 changes the saturation of a pixel to a mid-value between the pixel's current color and the specified color.

NOTE: *The `Saturation` parameter only functions when the `Blend` parameter is set to "colorize."*

`PreserveAlpha` - if set to "true", preserves the alpha channel of the target image as the alpha channel of the resulting image. The default is false.

`Blend` - specifies the type of blending used to combine the drawn object with the images. Blend options are: "Normal", "Darken", "Lighten", "Hue", "Saturation", "Color", "Luminosity", "Multiply", "Screen", "Dissolve", "Overlay," "HardLight", "SoftLight", "Difference", "Exclusion", "Dodge", "ColorBurn", "Under", "Colorize" (causes only the hue component of the source to be stamped down on the image), and "Prenormal".

NOTE: *"Burn" has been deprecated. "ColorBurn" results in the same blend.*

`Width` - specifies the thickness (in pixels) of the line that describes the rectangle. The default is 1.

NOTE: *If the `Fill` parameter is set to true, `Width` is ignored.*

`Smooth` - if set to "true", makes the edges of the rectangle smooth, preventing a pixellated effect. The default is false.

`Fill` - fills in the rectangle with the color specified by the `Color` or `Index` parameter. The default is false.

Example

```
var image = new Media();
image.load(name @ "family2.jpg");
image.rectangle(x @ 45, y @ 55, xs @ 283, ys @ 157, width @ 3);
image.save(type @ "jpeg");
```

reduce()

Applies a specified or generated color palette to the image. By default, this function generates an optimal palette of 256 colors.

NOTE: *MediaRich also supports Adobe Color Table (.act) files.*

Syntax

```
reduce(
       [Netscape @ <true, false>]
       [BW @ <true, false>]
       [LowMem @ <true, false>]
       [Pad @ <true, false>]
       [PreserveBackground @ <true, false>]
       [NoWarp @ <true, false>]
       [Name @ <"Palettes/filename.pal",
"virtualfilesystem:/filename.pal">]
       [Colors @ <1 to 256>]
       [Dither @ <value 0..10>]
       [DitherTop @ <value 0..10>]
       [layers @ <"layer list">] // (PSD files only)
       );
```

Parameters

`Netscape` - if set to "true", applies the Netscape default palette as an alternative to applying the default custom palette.

`BW` - if set to "true", applies the two-color, black and white palette.

`Pad` - ensures that the palette always contains the required number of colors. In a situation where there are fewer unique colors in the image than required for the palette, the extra colors are padded with black. If pad is not specified, the palette will shrink down to the number of unique colors available.

`PreserveBackground` - when dithering is used, eliminates any pixels in the source image that match the background color from the dithering process in the destination image. This can be used to eliminate fuzzy edges for an object against a solid color background.

`Nowarp` - if set to "true", turns off the normal color space warping that occurs when searching for a closest color to take into account the bias in the human eye. This is useful when reducing an image to an existing palette with a small number of colors, such as the Windows 16-color palette.

`Name` - specifies a palette file as the palette to be applied to the image. The following color palette files are installed on MediaRich:

- `128_Grays.pal`
- `16_Grays.pal`
- `256_Grays.pal`
- `32_Grays.pal`
- `4_Grays.pal`
- `64_Grays.pal`
- `8_Grays.pal`
- `Macintosh_16.pal`
- `Macintosh_256.pal`
- `Netscape.pal`
- `Windows_16.pal`
- `Windows_256.pal`

The default location for palette files is:

```
MediaRich/Shared/Originals/Media/Palettes
```

You can store additional palette files in this directory and use the `Name` parameter to specify the palette to be applied to the image.

NOTE: *You can modify the MediaRich server's `local.properties` file to change the default Media/Palettes directory. See the MediaRich Administrator's Guide for more information.*

MediaRich also allows you to set up virtual file systems and then use the `Name` parameter to load palettes from a virtual file system. Virtual file systems are defined in the MediaRich server's `local.properties` file. For example, if you define "MyPalettes:" to represent the path "C:/PALS/MyPalettes/" in the `local.properties` file, you can use files from the MyPalettes directory with the reduce() function:

```
image.reduce(name @ "MyPalettes:/custom.pal");
```

NOTE: *You may need to experiment with dithering and dithertop levels to achieve the results you want in the palette you use. For example, palettes with a bit-depth between 128 and 256 seem to appear best with a Dither value of 8 and a Dithertop value of 6.*

```
image.reduce(Name @ "Palettes/Windows_256.pal");
```

`Colors` - specifies the number of palette colors to be generated and applied. In the case of a media with multiple frames, all the frames are reduced to one palette based on the contents of all the frames.

NOTE: *The `Notbackcolor` parameter has been deprecated.*

`Dither` - determines the level of dithering to use for remapping image pixels to the palette. The default is zero, which is no dithering. While the dither value ranges from 0 to 10, the actual effects of different values vary according to the number of colors in the palette and their spread relative to each other.

`Dithertop` - if set to "true" when dithering is used, sets an upper threshold of how far the dithering algorithm will go to pick a color in order to correct color balance. The default value is 10. When an optimal (custom) palette is used, lowering the value of dithertop tends to reduce the pixelization of the image, making the dithering effect softer.

`layers` - for PSD files, specifies the layers to be affected. The layer numbers begin at 0 (background) and go up. For more information see "load()" on page 101.

Example

```
var image = new Media();
image.load(name @ "car.tga");
image.reduce(colors @ 256, dither @ 3, pad @ true);
image.save(type @ "jpeg");
```

rotate()

Rotates the media by the specified angle in degrees. This function fully supports the CMYK color-space.

Syntax

```
rotate(
        Angle @ <value 0 to infinity>
        [ ResizeCanvas @ <true, false>]
        [ Smooth @ <true, false>]
        [ Xs @ <pixels>]
        [ Ys @ <pixels>]
        [ layers @ <"layer list">] // (PSD files only)
        );
```

Parameters

`Angle` - specifies the number of degrees the image will be rotated. Positive numbers rotate clockwise and negative numbers rotate counter-clockwise.

`ResizeCanvas` - provides for the canvas of the image to be automatically enlarged in order to encompass the rotated image. The additional area uses the image's background color. For more information about setting an image's background color, see setColor().

NOTE: *The `Enlarge` parameter has been deprecated.*

`Smooth` - provides for smooth edges when rotating to something other than right angles.

`Xs` and `Ys` - specify how the image will be cropped after it is rotated.

`layers` - for PSD files, specifies the layers to be affected. The layer numbers begin at 0 (background) and go up. For more information see "load()" on page 101.

Example

```
var image = new Media();
image.load(name @ "pasta.tga");
image.rotate(angle @ 45, smooth @ true);
image.save(type @ "jpeg");
```

rotate3d()

Rotates the image in 3D along either the x-axis or y-axis. A positive angle rotates away from the viewer about the top or left edge, a negative angle rotates away from the viewer about the bottom or right edge.

This function fully supports media objects within the CMYK color-space.

Syntax

```
rotate3d(
        [alg @ <"Fast, Smooth, Best">]
        [anglex @ <angle ±89>]
        [angley @ <angle ±89>]
        [distance @ <value>]
        [layers @ <"layer list">] // (PSD files only)
        );
```

Parameters

`alg` - specifies the algorithm that will be used. The default algorithm is *fast*. The effect of the *best* algorithm is most apparent when scaling upward -- it uses a spline algorithm, giving superior results, but is slower than both the fast and smooth algorithms.

`anglex` - specifies the number of degrees the image will be rotated around the x axis. A positive angle rotates away from the viewer about the top or left edge. A negative angle rotates away from the viewer about the bottom or right edge.

`angley` - specifies the number of degrees the image will be rotated around the y axis. A positive angle rotates away from the viewer about the top or left edge. A negative angle rotates away from the viewer about the bottom or right edge.

NOTE: *You can specify a value for only one of the* `Anglex` *or* `Angley` *parameters, and only values between -89 and +89 are permitted.)*

`distance` - gives the distance in pixels the viewer is away from the image. The default value is twice the longest dimension of the image (which gives a nice look). If a more extreme perspective is required, use a smaller value for distance. If a less extreme perspective is required, use a larger value.

NOTE: *The value of* `distance` *must be greater than zero.*

`layers` - for PSD files, specifies the layers to be affected. The layer numbers begin at 0 (background) and go up. For more information see “load()” on page 101.

Example

```
var image = new Media();
image.load(name @ "pasta.tga");
image.rotate3d(angley @ 30, distance @ 28);
image.save(type @ "jpeg");
```

save()

Saves a Media object to the specified file. You can save the Media object as a BMP, EPS, GIF, JPEG, PCX, PDF, PICT, PNG, PPM, PSD, SWF, TIFF, TGA, or WBMP.

Syntax

```
save(
      [name @ "ftp://username:password@ftp.server_name/../
filename", "virtualfilesystem:/filename"]
      [type @ <"typename">]
      [embedICCProfile @ <true, false>]
      [interlaced @ <true, false>] // (GIF and PNG files only)
      [loopcount @ <value>] // (GIF files only)
      [removeduplicates @ <true, false>] // (GIF files only)
      [delay @ <value>] // (GIF files only)
      [disposalmethod @ <"mode">] // (GIF files only)
      [quality @ <value 1..100>] // (JPEG files only)
      [progressive @ <true, false>] // (JPEG files only)
      [baseline @ <true, false>] // (JPEG files only)
      [colorspace @ <"type">] // (JPEG files only)
      [highdetail @ <true, false>] // (JPEG files only)
      [dontoptimize @ <true, false>] // (JPEG files only)
      [compressionlevel @ <value 0..9>] // (PNG files only)
      [endian @ <"byte order">] // (TIFF files only)
      [compression @ <"mode">] // (TIFF files only)
      [SaveMetadata <true, false>]
      );
```

Parameters

`name` - specifies the name used to save a file to an FTP URL.

```
image.save(name@"ftp://jdoe:ax24z87@ftp.equilibrium.com/ test.gif");
```

NOTE: *FTP support is disabled by default for security reasons. Contact your MediaRich Administrator to enable this functionality.*

```
image.save(name@"ftp://jdoe:ax24z87@ftp.equilibrium.com/ test.gif");
```

`type` - specifies the file type of the saved image; otherwise, the type is derived from the extension of the file name. Valid type names are: "bmp", "eps", "gif", "jpeg", "pcx", "pict", "png", "ppm", "psd", "swf", "tiff", "targa", and "wbmp".

Saving an image as an SWF file creates a single-frame animation that can then be imported into a Flash movie.

NOTE: *The following formats support saving in the CMYK color-space: ".eps", ".psd", ".tif", and ".jpg".*

`embedICCProfile` - if set to "true", indicates that any destination profile associated with the image be embedded if the file format supports this.

NOTE: *The `save()` function supports ICC profiles for EPS, JPEG, PSD and TIFF files. However, `save()` has been deprecated and no longer does any color conversion. `ColorCorrect()` must now be used explicitly for converting from RGB to CMYK (or visa versa). Parameters for Color Profile Specifications (srcProfile, destProfile, and intent) are no longer supported by `save()`.*

`SaveMetadata` - if specified as "true", any metadata associated with the image will be embedded in the image. If the target file format does not support metadata, this parameter has no effect. For more information about MediaRich's metadata support, see Appendix A, "MediaRich Metadata Support" on page 169.

Additional Parameters for GIF Files

`Loopcount` - sets the number of times the frames plays after loading. The default is 0 (infinite looping).

`Removeduplicates` - if set to "true", causes the GIF writer to remove duplicate frames and combine their delay times into a single frame. The default is false.

Delay - sets the delay time (in hundredths of a second) for all frames in the GIF, overriding any values that are stored in each frame.

`DisposalMethod` - indicates the mode of compression used when saving the GIF. The possible modes are:

- Auto - this is the default behavior if no option is specified. It determines the best compression using all of the GIF specification features.
- Compatible - this mode sets compatibility for Netscape and Opera browsers. The GIF writer still automatically calculates delta rectangles for each frame and does transparent color compression, but replaces any "restore-to-previous" instructions in the GIF with "restore with background".

NOTE: *The Compatible mode may result in a less efficient GIF, depending on how the pixels are laid out in each frame. There will be no difference if the GIF is not animated and has no transparent areas that are visible down to the browser's background.*

- ManualUnspecified - this mode disables any compression to allow compatibility with any applications that do properly follow the GIF specification.
- ManualLeave - this mode prevents the disposal of the preceding frame when displaying the current frame.
- ManualUseBG - this mode replaces the preceding frame with the background color - usually transparent - when displaying the current frame.
- ManualUsePrev - this mode restores the preceding frame before displaying the next frame.

NOTE: *If the original image has more than 256 colors, you must apply the* `reduce()` *function before the* `save()` *function.*

Additional Parameters for GIF and PNG Files:

`Interlaced` - if set to "true", turns graphic interlacing on. The default is false.

Additional Parameters for JPEG Files

`Quality` - sets the level of quality on a scale from 0 to 100. The default is 85.

`Progressive` - if set to "true", allows browsers to load the image in stages. The default is false.

`Baseline` - if set to "true", saves the JPEG using the optimized baseline format. The default is false, or the standard baseline format.

`Colorspace` - specifies the colorspace format in which the JPEG is saved. The default is "Std". Other valid colorspace format options are:

- "Gray"
- "RGB"
- "YUV"
- "CMYK"
- "YCCK" (usually compresses better when saving CMYK data)

`Highdetail` - if set to "true", improves overall image quality. The default is false, unless the `Quality` parameter is set to 100, in which case it is automatically enabled.

NOTE: *This option yields better results with drawings than photographs.*

`Dontoptimize` - disables the optimize feature of the JPEG writer. The default is false.

Additional Parameter for PNG Files

`Compressionlevel` - sets compression level. The default is 6. (A `Compressionlevel` value of "9" can be very slow in processing.)

Additional Parameters for EPS Files

`binary` – if set to "false", writes an ASCII EPS file. If not specified, it defaults to "true" and writes a binary EPS file.

`preview` – if set to "false", writes an EPS file without a TIFF preview. If not specified, it defaults to "true" and writes a preview TIFF to the file.

`debabCompat` – if set to "false", writes a smaller file that is not compatible with DeBabelizer. If not specified, it defaults to "true" and writes larger, but DeBabelizer compatible file.

previewAtEnd – if set to “true”, writes the TIFF preview at the end of the file. If not specified, it defaults to “false” and writes the TIFF preview at the beginning of the file.

Additional Parameter for TIFF Files

endian - indicates the byte order. Values of “big”, “mac”, “motorola”, and “sparc” save the data in big-endian byte order. “little”, “pc”, “intel”, and “x86” save the data in little-endian byte order.

Compression - indicates the compression scheme to use. Valid values are:

- none
- rle - compresses runs of identical byte sequence values into code only a few bytes in length
- faxg3 - outputs TIFF files that conform to the Group 3 FAX format.
- faxg4 - outputs TIFF files that conform to the Group 4 FAX format.
- jpg - outputs TIFF files using the JPEG compression scheme.

NOTE: *When the JPG compression scheme is specified, a CompressionQuality parameter can optionally be supplied in the save() function. CompressionQuality sets the level of quality on a scale from 0 to 100. This parameter controls the quality vs. compression ratio — high values produce large files with better quality and lower values produce smaller files with poorer quality.*

- lzw - outputs a lossless, dictionary-based compression, which results in fair compression ratios (for most images, it produces a compression ratio of about 2:1).
- packbits - uses a Run-Length Encoded (“RLE”) compression.

Example

```
var image = new Media();
image.load(name @ "peppers.tif");
image.save(type @ "gif", interlaced @ true, loopcount @ 100,
removeduplicates @ true, delay @ 400);
```

saveEmbeddedProfile()

Saves the profile embedded in an image to the disk file specified.

Syntax

```
saveEmbeddedProfile(
        name @ <"filename.icc">
        );
```

Parameters

name - specifies the name of the file where the profile is to be stored. The string must be in quotes. If the string does not specify a file system the default is the color file system. See “File Systems” on page 14 for more information.

scale()

Scales the image to the specified size. This function fully supports the CMYK color-space.

Syntax

```
scale(
      [Alg @ <"Fast", "Smooth", "Outline", "Best">]
      [Constrain @ <true, false>]
      [Xs @ <pixels>, <percentage + "%">]
      [Ys @ <pixels>, <percentage + "%">]
      [X1 @ <pixels>]
      [Y1 @ <pixels>]
      [X2 @ <pixels>]
      [Y2 @ <pixels>]
      [PreserveBackground @ <true, false>]
      [PreserveBackgroundCutoff @ <value 0..100>]
      [PadColor @ <color in hexadecimal or rgb>]
      [PadIndex @ <value 0..16777215>]
      [Transparency @ <value 0..255>]
      [TransparentCutoff <-1, 0..255>]
      );
```

Parameters

`Alg` - specifies the algorithm that will be used. The default algorithm is *fast*. The *outline* algorithm is designed for black and white images only. The effect of the *best* algorithm is most apparent when scaling upward -- it uses a spline algorithm, giving superior results, but is slower than both the fast and smooth algorithms.

`Constrain` - specifies that the ratio between xs and ys is maintained relative to the original image. If `Xs` and `Ys` values are specified and constrain is set to true, the image size will be padded to preserve the aspect ratio of the source. If the padColor parameter is not set then the padcolor is determined by the backcolor.

`Xs` and `Ys` - specify the size of the generated image, either as an absolute (in pixels), or as a percentage of the selection in the original. Use `X1`, `Y1`, `X2`, and `Y2` to specify the selected area. If no area is selected, the percentage is based on the original image size.

NOTE: *Putting a percentage sign after the number signifies a percentage. Where either xs or ys is not specified, the original dimension is assumed.*

`X1` and `Y1` - represent the upper left corner of the area to be scaled. The default is the original image's upper left corner.

`X2` and `Y2` - represent the lower right corner of the area to be scaled. The default is the original image's lower right corner.

`PreserveBackground` - when scaling an image that contains an object surrounded by a solid background color, setting this parameter to "true" avoids anti-aliasing the edge of the object with the background. Anti-aliasing is a method of eliminating jagged edges by blending pixel colors with the background. When working with an object on a solid background, however, most users find it preferable to maintain a sharp, clean edge, because the blending can often produce an undesired halo effect.

`PreserveBackgroundCutoff` - specifies the threshold for `PreserveBackground`. The default threshold percentage is 67%, which means that the background color will be preserved unless 67% or more of the pixels use the background color.

`Padcolor` or `Padindex` - specifies the color to be used where the new image dimensions extend beyond the current image. If a pad color is not specified, the image's background color is used by default. For more information about setting an image's background color, see setColor().

`Transparency` - specifies the transparency (255 is opaque and 0 is transparent) of the padded area's alpha channel. This parameter is useful when the cropped image is used in a composite().

NOTE: *If the cropped image is not 32-bit before cropping, the transparency information is not used on the next composite() function.*

`TransparentCutoff` - specifies a value that controls the selection of the transparent pixel when scaling images with color palette. If the scaled alpha channel value is less than or equal to the `transparentCutoff` value, the transparent pixel is selected. A value of -1 (default) ignores the scaled alpha value and performs the normal reverse color lookup.

Example

```
var image = new Media();
image.load(name @ "pasta.tga");
image.scale(xs @ "75%", constrain @ true);
image.save(type @ "jpeg");
```

selection()

Creates a selection from the specified Media object.

The selected area can be thought of as a grayscale image or alpha channel that determines the way in which a given transform is applied to an image. Where the selection is 255, the transform or function is applied to the image pixel; where the selection is zero, the transform is not applied. In cases where the selection is between 1 and 254, the transform is applied to the source pixel, and the result is then blended with the original pixel based on the selection value. This function also fully supports the CMYK color-space.

NOTE: *If using with two source images, both images must be the same size. This can be accomplished with the scale(), getHeight(), or getWidth() function.*

This function can be used in conjunction with the following functions: adjustHsb(), adjustRgb(), blur(), blurBlur(), blurGaussianBlur(), blurMoreBlur(), blurMotionBlur(), colorize(), composite(), equalize(), noiseAddNoise(), otherHighPass(), otherMaximum(), otherMinimum(), pixellateMosaic(), pixellateFragment(), sharpenSharpen(), sharpenSharpenMore(), sharpenUnsharpMask(), stylizeDiffuse(), stylizeEmboss(), stylizeFindEdges(), and stylizeTraceContour().

Syntax

```
selection(
      [ Source @ <user-defined Media object name>]
      [ Fill @ <value 0..255)]
      [ X @ <pixel>]
      [ Y @ <pixel]
      [ BackColor @ <true, false>]
      [ Color @ <color in hexadecimal or rgb>]
      [ Index @ <value 0..16777215>]
      [ ColorType @ <"Cyans", "Magentas", "Yellows", "Reds",
"Greens", "Blues", "Hilites", "Midtones", "Shadows">]
      [ Invert @ <true, false>]
      [ Remove @ <true, false>]
      [ Opacity @ <value 0..255>]
      [ Radius @ <value 1..600>]
      [ layers @ <"layer list">] // (PSD files only)
      );
```

Parameters

You specify the area of selection using one of two sets of parameters:

1 Using `Source` - When you use `Source`, the system interprets the image as a grayscale (if it is not one). Loading a selection replaces one that is already active.

NOTE: *The `Name` parameter has been deprecated.*

- Before creating a new selection, you must load() the image. Then, use the `Source` parameter to refer to that image by its user-defined Media object name.
- If the source and target images are of different size, use the `Fill` parameter to specify what value pixels have in the selection mask that fall outside the size of the selection image. The default is 0.
- The `X` parameter determines at what horizontal position the top-left corner of the source image is placed on the target image. If the X parameter is not specified, the selection image will be centered over the target image horizontally.
- The Y parameter determines at what vertical position the top-left corner of the source image is placed on the target image. If the Y parameter is not specified, the selection image will be centered over the target image vertically.
- Using Backcolor, Color, Index, or ColorType- Use one of these parameters to create a selection from an image that includes all pixels that match the specified color or color type.

The color can be specified as the background color, or as all pixels of a specified color, index value, or color type. In the event that a selection containing everything except a particular color is required, the "invert" parameter can be added to the command.

`Invert` - reverses the opacity values of the current selection (for example, 0->255 and 255->0).

NOTE: *If the invert parameter is used, it will invert both the opacity and the backcolor, color, and index values. If you desire to invert one but not the other, you will need to write separate commands.*

`Remove` - de-activates any current selection.

`Opacity` - alters the current level of transparency for the selection. Applying an opacity level of 128 will increase the transparency level of the selection by 50%. If reduced, the level of the selection cannot be increased again.

`Radius` - when the backcolor, color, index, or color type parameter is also specified, this parameter selects all pixels of colors most similar to the specified color (using the specified color as the starting point) and increases the range of similar colors included in the selection as the value for `Radius` increases. The value for this parameter must be higher than zero. For example:

```
image.selection(Color @ 0x008000, Radius @ 20);
```

This example will create a selection consisting of all the colors in the image that are most similar to this color green within a radius of 20.

`layers` - for PSD files, specifies the layers to be included. The layer numbers begin at 0 (background) and go up. For more information see "load()" on page 101.

Example

```
var Target = new Media();
var Source = new Media();
Target.load(name @ "peppers.tga");
Source.load(name @ "Bears.tga");
Target.selection(source @ Source, opacity @ 240);
Target.adjustHsb(hue @ 75, saturation @ 75);
Target.save(type @ "jpeg");
```

setColor()

Sets the background color, foreground color, and transparency state of an image. Very few formats support saving of this information, so this function is primarily used in internal calculations in conjunction with other functions such as `arc()` and `drawText()` and supports the CMYK color-space.

When an image is initially loaded into memory, the foreground and background colors are initialized according to the following order of precedence:

- For indexed images:

Background color will be index 0.

Foreground color will be the last indexed color.

- For all other images:

Background color will be black.

Foreground color will be white.

NOTE: *If the image's file type supports them and its background, transparency and/or foreground colors have been set, those values will be used.*

Unless specifically changed, the initial values will be retained and used throughout all subsequent transformations. To be sure of the values used, it is best to use specific settings.

Syntax

```
setColor(
      [ BackColor @ <color in hexadecimal or rgb>]
      [ ForeColor @ <color in hexadecimal or rgb>]
      [ BackIndex @ <value 0..16777215>]
      [ ForeIndex @ <value 0..16777215>]
      [ Transparency @ (true, false)]
      [ Popular @ (true, false)]
      [ Precise <true, false>]
      [ layers @ <"layer list">] // (PSD files only)
      );
```

Parameters

`Backcolor` - specifies the background color as a specific RGB or hexidecimal value.

`Forecolor` - specifies the foreground color as a specific RGB or hexidecimal value.

`Backindex` - specifies the background color as an index value. Direct indexing is primarily used for indexed images, but can be used for any image type to select a specific pixel value.

`Foreindex` - specifies the foreground color as an index value.

`Transparency` - if this parameter is set to "false", the whole image is considered opaque. If set to "true", the pixels in the image that match the background color are considered transparent. Transparency is typically used when generating an alpha channel for an image (such as compositing an image that is not 32-bit). Transparency is also supported when saving to the GIF format and, if 8-bit or less, to the PNG format.

`Popular` - if set to "true", finds the most popular color or index in the image. For images above 16-bit color depth, the image is processed at 18-bit resolution.

NOTE: *The `Popular` parameter overrides any settings specified by the `Backcolor`, `Forecolor`, `Backindex` or `Foreindex` parameter. In addition, this parameter does not support the CMYK color-space.*

`Precise` - If `Popular` is specified and this is set to "true", the method uses precision in the calculation of the most popular color. If set to false (default), the color returned will be a close approximation of the actual color that appears most often in the image.

`layers` - for PSD files, specifies the layers to be included. The layer numbers begin at 0 (background) and go up. For more information see "load()" on page 101.

Example

```
var image = new Media();
image.load(name @ "car.tga");
image.setColor(backcolor @ 0xC2270B);
image.crop(alg @ "backcolor");
image.save(type @ "jpeg", compressed @ true);
```

setFrame()

Replaces the Media object for the specified frame (if available). Can be used with getFrame() to modify an animation.

Syntax

```
<object name>.setFrame(
        <frame offset>
        <source Media object name>
);
```

Parameters

The frame offset (starting from 1) specifies which frame in the target Media object gets replaced by the named source Media object (which consists of a single frame).

Example

```
var image = new Media();
image.load(name @ "Images/clock.gif");   // Load an animated GIF with
four frames
image2 = image.getFrame(2);
image2.flip(axis @ "Vertical");
image.setFrame(2, image2);
image.save(name @ "newclock.gif");
```

setLayer()

Replaces the Media object for the specified layer (if available). Used in conjunction with getLayer(), this is commonly used to modify layer contents before calling the collapse() function.

Syntax

```
<object name>.setLayer(
        <layer index, object name>
        );
```

Parameters

The first parameter represents the layer index (the first layer in a file is layer 0).

The second parameter names the Media object that contains the data with which the layer is replaced.

Example

```
var image = new Media();
var Layer = new Media();
Layer = image.getLayer(2);
Layer.rotate(angle @ 30);
image.setLayer(2, Layer);
```

setLayerBlend()

Sets the blending mode of the media layer with the specified index (if available).

Syntax

```
<object name>.setLayerBlend(
        <layer index>
        <"blending mode">
        );
```

Parameters

The first parameter specifies the layer index (starting from 0).

The second parameter specifies the blending mode to be used. Blend options are: "Normal", "Darken", "Lighten", "Hue", "Saturation", "Color", "Luminosity", "Multiply", "Screen", "Dissolve", "Overlay," "HardLight", "SoftLight", "Difference", "Exclusion", "Dodge", "ColorBurn", "Under", and "Colorize" (causes only the hue component of the source to be stamped down on the image).

NOTE: *"Burn" has been deprecated. "ColorBurn" results in the same blend.*

Example

```
var image = new Media();
image.setLayerBlend(2, "Difference");
```

setLayerEnabled()

Sets the specified layer as either enabled or disabled. If you use the `collapse()` function without naming specific layers, MediaRich collapses all enabled layers and ignores disabled layers. Use the `setLayerEnabled()` function or the eye icon in Photoshop to enable/disable a layer. Use the getLayerEnabled() function to determine if a layer is enabled or not.

Syntax

```
<object name>.setLayerEnabled(
        <layer index>,
        <true, false>
);
```

Parameters

The first parameter specifies the desired layer index (starting from zero).

If setLayerEnabled is set to true, the layer is enabled; if set to false, the layer is disabled.

Example

```
if (image.getLayerEnabled(2) == false)
image.setLayerEnabled(2, true);
...}
```

setLayerHandleX()

Sets the HandleX of the media layer with the specified index (if available).

Syntax

```
<object name>.setLayerHandleX(
        <layer index>
        <"position">
        );
```

Parameters

The first parameter specifies the desired layer index (starting from zero).

The second parameter sets the selected layer's attachment point on the *x*-axis. The default is "Center." Other options are "Left" and "Right".

Example

```
var image = new Media();
image.setLayerHandleX(2, "Right");
```

setLayerHandleY()

Sets the HandleY of the media layer with the specified index (if available).

Syntax

```
<object name>.setLayerHandleY(
        <layer index>
        <"position">
        );
```

Parameters

The first parameter specifies the desired layer index (starting from zero).

The second parameter sets the selected layer's attachment point on the *x*-axis. The default is "Middle". Other options are "Top" and "Bottom".

Example

```
var image = new Media();
image.setLayerHandleY(2, "Bottom");
```

setLayerOpacity()

Sets the opacity of the media layer with the specified index (if available).

Syntax

```
<object name>.setLayerOpacity(
        <layer index>
        <value 0..255>
        );
```

Parameters

The first parameter specifies the desired layer index (starting from zero).

The second parameter specifies opacity of the selected layer, with a value of 255 indicating completely solid.

Example

```
var image = new Media();
image.setLayerOpacity(2, 128);
```

setLayerPixels()

Replaces the pixel data in a named layer of the target Media object with the pixel data from a layer in the source Media object. Any attributes associated with the target layer are preserved.

Syntax

```
<object name>.setLayerPixels(
        <layer index>,
        <Media object>
        );
```

Parameters

The first parameter specifies the layer in the target image that gets its pixels replaced. The default is the first layer (starting from zero).

The second parameter specifies the source Media object. Before you can use `setLayerPixels()` you must `load()` the source image. If the source image has multiple layers, the first one is used.

Example

```
var Target = new Media();
var Source = new Media();
Target.load (name @ "banner.psd");
Source.load (name @ "fishes.psd");
Target.setLayerPixels(3,Source);
Target.save(type @ "jpeg");
```

setLayerX()

Sets the X composite offset of the media layer with the specified index (if available).

Syntax

```
<object name>.setLayerX(
       <layer index>
       <position>
       );
```

Parameters

The first parameter specifies the desired layer index (starting from zero).

The second parameter specifies the position of the selected layer along the *x*-axis of the composite image, with the layer's center point used as the anchor point. For example, a value of 50 positions the center point at pixel 50 on the *x*-axis of the composite image.

Example

```
var image = new Media();
image.setLayerX(2, 50);
```

setLayerY()

Sets the Y composite offset of the media layer with the specified index (if available).

Syntax

```
<object name>.setLayerY(
       <layer index>
       <position>
       );
```

Parameters

The first parameter specifies the desired layer index (starting from zero).

The second parameter specifies the position of the selected layer along the *y*-axis of the composite image, with the layer's center point used as the anchor point. For example, a value of 100 positions the center point at pixel 100 on the *y*-axis of the composite image.

Example

```
var image = new Media();
image.setLayerY(2, 100);
```

setLayerPixels()

Sets the image in the specified layer of the current media to the specified raster.

Syntax

```
<object name>.setLayerPixels(
        <layer index>
        <media>
        );
```

Parameters

`layerIndex` - the index of the layer to be replaced. This must exist in the current image.

`media` - the raster to use as the replacement. This must consist of a single raster.

setMetadata()

Attaches metadata specified by data to the image for the specified format. If data is not specified or null, clear any metadata for the specified format. Data must be an appropriate XML document for the format specified and should be validated against the relevant schema.

NOTE: *For Exif and IPTC formats, only elements present in the respective schema will be added to the image on save. Any other data present in the document will be ignored.*

Syntax

```
<object name>.setMetadata(
        <"format">,
        <"data">
        );
```

Parameters

`format` - the format of the document specified by data. Valid values are "Exif", "IPTC", and "XMP".

`data` - a string containing an XML document corresponding to format.

setPixel()

Sets the color of one pixel to the chosen color value. This works in both RGB and CMYK color spaces.

Syntax

```
<objectname>.setPixel(
        [x @ <"pixel">]
        [y @ <"pixel">]
        [transparency @ 0-255, or true or false]
        [color @ <"color">]
        [layers @ <"layer list">] // (PSD files only)
        );
```

Parameters:

`x` and `y` - required parameters that specify the coordinates of the target pixel. The top-left corner of an image is represented by the coordinates 0,0.

`transparency` - this optional parameter sets the alpha channel of the pixel to that value. Valid values are 0-255. If this parameter is not specified, the alpha channel (if any) of the original image remains unchanged.

`color` - this optional parameter specifies the color that will replace the designated pixel. The default value for color is the image's foreground color. For more information about setting an image's foreground color, see "setColor()" on page 131.

`layers` - for PSD files, specifies the layers to be affected. The layer numbers begin at 0 (background). For more information see "load()" on page 101

The foreground color may vary with this function, depending on the original Media object. If the object has a set foreground color, or it is set with the setColor() function, MediaRich uses the set color. However, if the object has no set foreground color, MediaRich does the following:

- For objects with 256 colors or less, MediaRich uses the last color index.
- For objects with 15-bit or greater resolution (including the CMYK color-space), MediaRich uses white.

Example

```
Image = new Media();
FindColor = "0x000000";
MakeColor = "0xff00ff";

Image.load(name @ "image/32bit.psd");
Rows = Image.getWidth();
Columns = Image.getHeight();
for (x = 0; x < Rows; x++)
{
        for (y = 0; y < Columns; y++)
        {
```

```
        if (Image.getPixel(x @ x, y @ y) == FindColor)
        {
        Image.setPixel(x @ x, y @ y, rgb @ MakeColor);
        }
        }
}
Image.save(type @ "jpeg")
```

setResolution()

Changes the DPI of the image in memory and fully supports media objects within the CMYK color-space.

NOTE: *This information can be stored only in the following formats: BMP, GIF, JPEG, PCT, PCX, PNG, PSD, and TIFF.*

Syntax

```
setResolution(
       [ Dpi @ <value 0.00 to 10,000.00>]
       [ XDpi @ <value 0.00 to 10,000.00>]
       [ YDpi @ <value 0.00 to 10,000.00>]
       [ layers @ <"layer list">] // (PSD files only)
       );
```

Parameters

`Dpi` - sets the resolution of the image in memory. The resolution value must be greater than 0, but may be decimal.

`Xdpi` and `Ydpi` - set the resolution on their respective axis only.

NOTE: *When the `Dpi` parameter and one or both of the single axis values are given, the axis value overrides the DPI value.*

`layers` - for PSD files, specifies the layers to be affected. The layer numbers begin at 0 (background) and go up. For more information see “load()” on page 101.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.setResolution(dpi @ 300, xdpi @ 200);
image.save(type @ "jpeg");
```

setSourceProfile()

Sets the embedded profile for an image to the specified source profile. This profile replaces any existing embedded profile for the image.

NOTE: *The colorspace of the specified source profile must match the colorspace of the image.*

Syntax

```
setSourceProfile(
        sourceProfile @ <"filename.icc">
          );
```

Parameters

`SourceProfile` - specifies the profile to use as the images new embedded profile. For more information about specifying profiles, see "colorCorrect()" on page 59.

For more information about color management please refer to Appendix B, "MediaRich Color Management."

sharpenSharpen()

Makes the edges in the image more pronounced.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
sharpenSharpen();
```

Parameters

This function has no parameters.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.sharpenSharpen();
image.save(type @ "jpeg");
```

sharpenSharpenMore()

Sharpens the clarity of an image, similar to the sharpenSharpen() function, but to a greater extent.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
sharpenSharpenMore();
```

Parameters

This function has no parameters.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.sharpenSharpenMore();
image.save(type @ "jpeg");
```

sharpenUnsharpMask()

Enhances the edges and details of an image by exaggerating the differences between the original image and a Gaussian-blurred version.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
sharpenUnsharpMask(
       [Radius @ <value 0.10..250>]
       [Amount @ <value 1..500>]
       [Threshold @ <value 1..255>]
       );
```

Parameters

`Radius` - specifies the extent of the blurring effect. The default is 1.

`Amount` - specifies the extent of the enhancing effect. The default is 50.

Threshold - specifies the degree to which a blurred version of a pixel must be different from the original version before the enhancement takes effect. The default is 0.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.sharpenUnsharpMask(Radius @ 18, Amount @ 450, Threshold @ 125);
image.save(type @ "jpeg");
```

sizeText()

Returns the width, height, and lines for the specified parameters. This function is available on Windows only.

Syntax

```
sizeText(
        [ text @ <"string">]
        [ font @ <"font family">]
        [ style @ <"modifier">]
        [ Size @ <value 1..4095>]
        [ justify @ <"justify">]
        [ Justify @ <"left", "center", "right", "justified">]
        [ spacing @ <"spacing">]
        [ Line @ <value 01. to 10>]
        [ smooth @ <true, false>]
        [ clearType @ <"clearType">]
        [ kern @ <true, false>]
        [ ClearType @ <true, false>] //(windows only)
);
```

Return Values

width - the overall width of the text in pixels.

height - the overall height of the text in pixels.

lines - the number of lines of text that will be drawn.

Parameters

`Font` - specifies the TrueType or PostScript font family name to be used, for example, "Arial". MediaRich supports Type 1 (.pfa and .pfb) PostScript fonts only.

NOTE: *The size of the font in pixels is dependent on the resolution of the resulting image. If the resolution of the image is not set (zero), the function uses a default value of 72 DPI.*

The default location for fonts specified in a MediaScript is the `fonts` file system. Which includes both the MediaRich `Shared\Originals\Fonts` folder and the default system fonts folder. If a MediaScript specifies an unavailable font, MediaRich generates an error.

NOTE: *You can modify the MediaRich server's `local.properties` file to change the default fonts directory. See the MediaRich Administrator's Guide for more information.*

`Style` - specifies the font style. You can use any combination of modifiers. Each modifier should be separated by a space character.

NOTE: *The `Style` parameter is not available if MediaRich is running on Mac, Linux, or Solaris.*

Weight modifiers modify the weight (thickness) of the font. Valid weight values, in order of increasing thickness, are:

- "thin"
- "extralight" or "ultralight"
- "light"
- "normal" or "regular"
- "medium"
- "semibold" or "demibold" ("semi" or "demi" are also acceptable)
- "bold"
- "extrabold" or "ultrabold" ("extra" or "ultra" are also acceptable)
- "heavy" or "black"

Other `Style` parameters are "Underline", "Italic" or "Italics", and "Strikethru" or "Strikeout").

NOTE: *You can combine `Style` parameters. For example: `Style @ "Bold Italic"`*

`Text` - specifies the text to be drawn. The text string must be enclosed in quotes. To indicate a line break, insert `\n` into the text.

`Size` - the point size of the font to be used. The default size is 12.

`Justify` - specifies how the text will be justified. The default is "center". Other options are "left", "right", and "justified". This parameter only affects text with multiple lines.

`Wrap` - if specified, uses the value to force a new line whenever the text gets longer than the specified number of pixels (in this case correct word breaking is used).

`Line` - specifies the line spacing. The default spacing between lines of text is 1.5.

`Smooth` - specifies that the text is drawn with five-level anti-aliasing.

`Spacing` - adjusts the spacing between the text characters. The default is 0. A negative value draws the text characters closer together.

`Kern` - if set to “true”, optimizes the spacing between text characters. By default this is set to true. If you do not want to use kerning, this must be specified as false.

NOTE: *PostScript fonts store the kerning information in a separate file with a .afm extension. This file must be present in order for kerning to be applied to the text.*

`ClearType` - if specified as “true”, the Windows ClearType text renderer will be used if available.

stylizeDiffuse()

Makes the image appear as though viewed through a soft diffusion filter, with options to lighten or darken the effect.

Syntax

```
stylizeDiffuse(
       [Radius @ <value 0..10000>]
       [Mode @ <"mode"]
       );
```

NOTE: *This function is “selection aware,” so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see “selection()” on page 129.*

Parameters

`Radius` - specifies the extent of the diffusion effect. The default is 1 (almost no effect).

`Mode` - indicates the diffusion mode, such as “Lighten” and “Darken”. The default is “Normal” (no lightening or darkening effect).

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.stylizeDiffuse(Radius @ 10, Mode @ "Lighten");
image.save(type @ "jpeg");
```

stylizeEmboss()

Makes the image appear as though embossed on paper.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
stylizeEmboss(
      [Height @ <value 1..10>]
      [Angle @ <value -360..360>]
      [Amount @ <value 1..500>]
      );
```

Parameters

`Height` - determines the depth of the embossing effect. The default is 3.

`Angle` - specifies the angle of the light source. The default is 135 (light source comes from the upper left).

`Amount` - specifies the extent of the effect; the higher the value, the greater the detail. The default is 100.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.stylizeEmboss(Height @ 2, Angle @ 90, Amount @ 250);
image.save(type @ "jpeg");
```

stylizeFindEdges()

Traces the edges (areas of significant transitions) of the image with broad lines.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
stylizeFindEdges(
        [ Threshold @ <value 0..255>]
        [ Grayscale @ <true, false>]
        [ Mono @ <true, false>]
        [ Invert @ <true, false>]
        );
```

Parameters

`Threshold` - specifies how sharp an edge must be to included. The default is 0.

`Grayscale` - produces a monochromatic result. The default is false.

`Mono` - when set to "true", causes all edges above the threshold value to default to 255. The default is false.

`Invert` - reverses the default foreground and background colors. The default is false.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.stylizeFindEdges(Threshold @ 125, Grayscale @ true, Mono @
true, Invert @ true);
image.save(type @ "jpeg");
```

stylizeTraceContour()

Creates a contour-line effect by locating the transitions of the more significant bright areas and outlining them for each color channel.

NOTE: *This function is "selection aware," so that if a selection has been made, the system applies it based on the current selection. For more information about making selections, see "selection()" on page 129.*

Syntax

```
stylizeTraceContour(
        [ Level @ <value 0..255>]
        [ Upper @ <true, false>]
        [ Invert @ <true, false>]
        );
```

Parameters

`Level` - indicates the level of each color gun. The default is 128.

`Upper` - if set to "true", causes the upper edge to be delineated. The default is false (the lower edge is delineated).

`Invert` - if set to "true", reverses the default foreground and background colors. The default is false.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.stylizeTraceContour(Level @ 96, Upper @ true, Invert @ true);
image.save(type @ "jpeg");
```

zoom()

Zooms in on a specified portion of the media and fits it to the specified size. This constitutes a crop followed by a scale. This function fully supports the CMYK color-space.

Syntax

```
zoom(
       [Alg @ <"Fast", "Smooth", "Outline", "Best">]
       [Xs @ <pixels>]
       [Ys @ <pixels>]
       [X @ <left pixel>]
       [Y @ <top pixel>]
       [Scale @ <value>]
       [PreserveBackground @ <true, false>]
       [PreserveBackgroundCutoff @ <value 0..100>]
       [PadColor @ <color in hexadecimal or rgb>]
       [PadIndex @ <value 0..16777215>]
       [Transparency @ <value 0..255>]
       [layers @ <"layer list">] // (PSD files only)
       );
```

Parameters

`Alg` - specifies the algorithm that will be used. The default algorithm is *fast*. The *outline* algorithm should be used for black and white images only.

`Xs` and `Ys` - specify the destination size of the media. The resulting image always fits these dimensions regardless of the scale and the source media.

`X` and `Y` - specify the position of the top-left corner of the region to be zoomed. These coordinates are specified relative to the destination media at scale @ 1.

`Scale` - specifies a real value and gives the magnification of the resulting image. For a value of "1", the whole media fits within the specified size.

`PreserveBackground` - when scaling an image that contains an object surrounded by a solid background color, setting this parameter to "true" avoids anti-aliasing the edge of the object with the background. Anti-aliasing is a method of eliminating jagged edges by blending pixel colors with the background. When working with an object on a solid background, however, most users find it preferable to maintain a sharp, clean edge, because the blending can often produce an undesired halo effect.

`PreserveBackgroundCutoff` - specifies the threshold for `PreserveBackground`. The default threshold percentage is 67%, which means that the background color will be preserved unless 67% or more of the pixels use the background color.

`Padcolor` or `Padindex` - specifies the color to be used where the new image dimensions extend beyond the current image. If a pad color is not specified, the image's background color is used by default. For more information about setting an image's background color, see setColor().

`Transparency` - specifies the transparency (255 is opaque and 0 is transparent) of the padded area's alpha channel. This parameter is useful when the cropped image is used in a composite().

NOTE: *If the cropped image is not 32-bit before cropping, the transparency information is not used on the next `composite()` function.*

`layers` - for PSD files, specifies the layers to be included. The layer numbers begin at 0 (background) and go up. For more information see "load()" on page 101.

Example

```
var image = new Media();
image.load(name @ "pasta.tga");
image.zoom(xs @ 100, ys @ 100, scale @ 2, x @ 20, y @ 30);
image.save(name @ "Result.tga");
```

XmlDocument Object

MediaRich allows users to interact with XML documents and supports all the objects, properties, and methods of the Document Object Model (DOM) Level 1 Core. The DOM Core is an application programming interface for XML documents. For information on using the DOM Level 1 Core objects, properties, and methods, see `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/`.

NOTE: *If your MediaScript uses the Xmldocument object, it must reference the xml.ms file that installs with MediaRich using the #include directive. Include the following line at the beginning of your script:*

```
#include "sys:xml.ms";
```

For more information about using the #include directive, see "Using the #include Directive" on page 12.

XmlDocument Object Properties

The XmlDocument object has all the properties of the DOM's Document class, as well as the `loaded` property.

loaded

A Boolean property, the value of which is determined by whether or not the XmlDocument is loaded.

Syntax

```
<object name>.loaded;
```

XmlDocument Object methods

The XmlDocument object has all the methods of the DOM's Document class, as well as the following methods:

- `new XmlDocument()` constructor
- `loadFile()`
- `loadString()`
- `save()`

new XmlDocument()

The XmlDocument object needs to be constructed using the `new XmlDocument()` constructor.

Syntax

```
var Test = new XmlDocument();
```

loadFile()

Loads an XML document from the file system.

Syntax

```
<object name>.loadFile(
        <"filename.xml", "virtualfilesystem:/filename.xml">
);
```

Parameters

The `loadFile()` function accepts an XML filename as it's only parameter. By default, MediaRich looks for XMLDocument files in the `write` file system which point to the following directory:

```
MediaRich/Shared/Originals/Media
```

NOTE: *You can modify the MediaRich server's `local.properties` file to change the default Media directory. See "File Systems" on page 14 for more information.*

MediaRich also allows you to set up virtual file systems and then load files from that location. Virtual file systems are defined in the MediaRich server's `local.properties` file. For example, if you define "XML:" to represent the path "`C:/010102/XML/`" in the `local.properties` file, you can use files from the XML directory with the `loadFile()` function:

```
XMLDoc.loadFile("XML:/customersUS.xml");
```

loadString()

Loads an XML file as a string, rather than as a file.

Syntax

```
<object name>.loadString(
        <"XML string">
);
```

Parameters

This function accepts an XML string as its only parameter. The string *must* include valid XML start and end tags.

Example

```
var test = new XMLDocument();
test.loadString("<html>sale</html>");
```

save()

Saves an XML document to the file system.

Syntax

```
<object name>.save(
        <"filename.xml", "virtualfilesystem:/filename.xml">
);
```

Parameters

This function accepts an XML filename as its only parameter. By default, MediaRich saves XMLDocument files in the `write` file system which point to the following directory:

```
MediaRich/Shared/Originals/Media
```

NOTE: *You can modify the MediaRich server's `local.properties` file to change the default Media directory. See "File Systems" on page 14 for more information.*

MediaRich also allows you to set up virtual file systems and then save files to that location. Virtual file systems are defined in the MediaRich server's `local.properties` file. For example, if you define "XML:" to represent the path "`C:/010102/XML/`" in the `local.properties` file, you can use files from the XML directory with the `save()` function:

```
XMLDoc.save("XML:/customersUS.xml");
```

The System Object

The System object provides access to system information and implements the following methods:

fontExists()

Returns true if a font with the specified name exists. It takes the font name as its only parameter.

getFontList()

Returns an array containing the names of all fonts known by the system.

Text Response Object

The TextResponse objects allow users to create response objects that take strings and save text files (“plain text”, “html”, or “xml”).

TextResponse Object Properties

There are no properties for this object.

TextResponse Object Methods

The TextResponse Object implements the following methods:

- `TextResponse()`
- `append()`
- `getText()`
- `getTextType()`
- `getType()`
- `setText()`
- `setTextType()`

TextResponse()

The TextResponse object needs to be constructed using the `new TextResponse()` constructor.

Syntax

```
var strObject = TextResponse(
        <textType>
        );
```

Parameters

`textType` - specifies the type of text using one of the following predefined values:

- `TextResponse.TypePlain` to save text as a plain text file with extension *.txt*.
- `TextResponse.TypeHtml` to save text as an HTML file with extension *.html*.
- `TextResponse.TypeXml` to save text as an XML file with extension *.xml*.

Example

```
#include "sys:/TextResponse.ms"
function main() {
        var strObject = new TextResponse(TextResponse.TypePlain);
        strObject.setText("FOO FOR YOU");
        strObject.append("\nAND FOO FOR ME");
        resp.setObject(strObject, RespType.Streamed);
}
```

append()

Appends the given text string to the text associated with the named object.

Syntax

```
<object name>.append(
       <"text string">
       );
```

Parameters

This function takes only a text string, which must be enclosed in quotation marks.

getText()

Returns the text associated with the named object.

Syntax

```
<object name>.getText();
```

Parameters

This function has no parameters.

getTextType()

Returns the text type associated with this object: “TextResponse.TypePlain”, “TextResponse.TypeHtml”, or “TextResponse.TypeXml”.

For more information about text types, see “TextResponse()” on page 153.

Syntax

```
<object name>.getTextType();
```

Parameters

This function has no parameters.

getType()

Returns the type of the named object, which is always “TextResponse”.

Syntax

```
<object name>.getType();
```

Parameters

This function has no parameters.

setText()

Sets the text string associated with this object.

Syntax

```
<object name>.setText(
       <"text string">
       );
```

Parameters

This function takes only a text string, which must be enclosed in quotation marks.

setTextType()

Sets the text type associated with this object. (must be one of the defines above).

Syntax

```
var strObject = TextResponse(
       <textType>
       );
```

Parameters

`textType` - specifies the text type using one of the following predefined values:

- `TypePlain` - saves text as a plain text file with extension *.txt*.
- `TypeHtml` - saves text as an HTML file with extension *.html*.
- `TypeXml` - saves text as an XML file with extension *.xml*.

ICC Profile Object

Dynamic enumeration of color profiles is provided by the IccProfile link library. The library allows clients to list profiles by colorspace and class or to query a specific icc profile.

static function list(colorspace, class)

Static method of the IccProfile class that returns an array of icc profile files corresponding to the specified colorspace and class.

The colorspace argument must be one or more of the following predefined constants, bitwise OR-ed (|) together:

- IccProfile.RGB
- IccProfile.CMYK
- IccProfile.LAB
- IccProfile.XYZ
- IccProfile.GRAY
- IccProfile.ALPHA
- IccProfile.PALETTE
- IccProfile.HLS
- IccProfile.HSV

The class argument must be one or more of the following predefined constants, bitwise OR-ed (|) together:

- IccProfile.MONITOR
- IccProfile.SCANNER
- IccProfile.PRINTER

- IccProfile.LINK
- IccProfile.ABSTRACT
- IccProfile.COLORSPACE
- IccProfile.NAMEDCOLOR
- IccProfile.UNKNOWN

Parameters

colorspace -> Bitwise OR (|) of desired colorspace values.

class -> Bitwise OR (|) of desired profile class values.

Example

The following code gets an array of all the ICC monitor and printer profiles for RGB and CMYK colorspaces.

```
#link "IccProfile.dll"

var profs = IccProfile.list(IccProfile.RGB | IccProfile.CMYK,
IccProfile.MONITOR | IccProfile.PRINTER);
```

ICC Profile Object Methods

The ICC Profile Object implements the following methods:

- IccProfile() Constructor
- getName()
- getPath()
- getClass()
- getColorspace()
- getConnectionspace()
- close()

NOTE: *For all paths in the IccProfile Object that do not specify a file system the default is the* `color` *file system. See "File Systems" on page 14 for more information.*

IccProfile()

Constructor for IccProfile object. Does not return a value.

Parameters

`Profile` - path to an ICC profile file, or a media object with an embedded profile.

NOTE: *The special profile names, "rgb" and "cmyk" may be used to denote the default RGB and CYMK profiles specified in the* `global.properties` *file under the keys "ColorManager.DefaultRGBProfile" and "ColorManager.DefaultCMYKProfile", respectively.*

Example

The following code creates an IccProfile object using the "USWebCoatedSWOP.icc" profile.

```
#link "IccProfile.dll"

var prof = new IccProfile("USWebCoatedSWOP.icc");
var image = new Media();
image.load(name @ "fileWithEmbeddedProfile.jpg");
var prof2 = new IccProfile(image);
```

getName()

Returns the friendly display name of the profile.

Parameters

This function takes no parameters.

Example

The following code creates an IccProfile object using the "USWebCoatedSWOP.icc" profile and gets the friendly name.

```
#link "IccProfile.dll"
var prof = new IccProfile("USWebCoatedSWOP.icc");
var name = prof.getName();
```

getPath()

Returns the path of the profile file.

Parameters

This method has no parameters.

Example

The following code creates an IccProfile object using the "USWebCoatedSWOP.icc" profile and gets the profile path.

```
#link "IccProfile.dll"
var prof = new IccProfile("USWebCoatedSWOP.icc");
var name = prof.getPath();
```

getClass()

Returns the profile class.

Parameters

This method has no parameters.

Example

The following code creates an IccProfile object using the "USWebCoatedSWOP.icc" profile and tests if it is a printer profile.

```
#link "IccProfile.dll"
var prof = new IccProfile("USWebCoatedSWOP.icc");
var isPrinter = prof.getClass() & IccProfile.PRINTER;
```

getColorspace()

Returns the profile colorpace.

Parameters

This function takes no parameters.

Example:

The following code creates an IccProfile object using the "USWebCoatedSWOP.icc" profile and tests if it is a CMYK profile.

```
#link "IccProfile.dll"
var prof = new IccProfile("USWebCoatedSWOP.icc");
var isCmyk = prof.getColorspace() & IccProfle.CMYK;
```

getConnectionspace()

Returns the Profile connection space as the jseNumber.

Parameters

This function takes no parameters.

Example

The following code creates an IccProfile object using the "USWebCoatedSWOP.icc" profile and gets the connectionspace.

```
#link "IccProfile.dll"
var prof = new IccProfile("USWebCoatedSWOP.icc");
var connectionspace = prof.getConnectionspace();
```

close()

Closes profile file. Returns no value.

Parameters

This function takes no parameters.

Example

The following code creates an IccProfile object using the "USWebCoatedSWOP.icc" profile and then closes the file:

```
#link "IccProfile.dll"
var prof = new IccProfile("USWebCoatedSWOP.icc");
prof.close();
```

IccProfile.dll

The following code returns text describing every available CMYK profile:

```
#link "IccProfile.dll"
#include "sys:/TextResponse.ms"

function main()
{
    var txt = new TextResponse();
   var profs = IccProfile.list(IccProfile.CMYK, IccProfile.UNKNOWN);
    for (var i = 0; i < profs.length; ++i)
    {
        txt.append(profs[ i]  + "\n");
        var currProf = new IccProfile(profs[ i] );
        txt.append("  name: " + currProf.getName() + "\n");
        txt.append("  path: " + currProf.getPath() + "\n");
        txt.append("  class: " + currProf.getClass() + "\n");
      txt.append("  colorspace: " + currProf.getColorspace() + "\n");
        txt.append("  connectionspace: " +
currProf.getConnectionspace() + "\n");
        currProf.close();
    }
    resp.setObject(txt, RespType.Streamed);
}
```

Zip Object

The Zip object is used to create and add files to a new zip archive.

Zip Profile Object Methods

The Zip Object implements the following methods:

- Zip(), Constructor
- addFile()
- save()

NOTE: *For all paths in the Zip Object that do not specify a file system the default is the* `write` *file system. See "File Systems" on page 14 for more information.*

Zip()

Constructor for Zip object.

Parameters

This function takes no parameters.

addFile()

Adds a file to the zip archive. Note that the file is not actually read until the save() method is called.

Parameters

`filePath` - specifies the VFS path of file to add to archive.

`archivePath` - specifies a full path of file as stored in archive.

save()

Creates a new zip archive and compresses all the files specified by calls to addFile().

Parameters

`archiveName` - specifies a path to new archive file.

Example

```
var myZip = new Zip();
myZip.addFile("images/image1.jpg", "image1.jpg");
myZip.addFile("images/image2.jpg", "image2.jpg");
myZip.save("zip/files.zip");
```

The Zip object can also be used as a response object. For example, the following code creates a zip archive as a cached response:

```
var myZip = new Zip();
myZip.addFile("images/image1.jpg", "image1.jpg");
myZip.addFile("images/image2.jpg", "image2.jpg");
resp.setObject(myZip);
```

Unzip Object

The Unzip object is used to extract files from an existing zip archive.

Unzip Profile Object Methods

The Unzip Object implements the following methods:

- Unzip() constructor
- open()
- extractAll()
- firstFile()
- nextFile()
- getFileName()
- extractFile()
- close()

NOTE: *For all paths in the UnZip Object that do not specify a file system the default is the* `read` *file system. See "File Systems" on page 14 for more information.*

Unzip()

Constructor for Unzip object.

Parameters

This function takes no parameters.

open()

Opens an existing archive. The archive will remain open until one of the following occurs: another archive is opened, the archive is explicitly closed with the close() method, or the Unzip object is garbage collected.

Parameters

`archiveName` - specifies the path to the existing archive file.

extractAll()

Extracts all files in the zip archive to the specified directory. The full paths stored in the archive will be preserved in the destination directory.

Parameters

`dir` - specifies the VFS path of the directory to extract files.

firstFile()

Resets file iterator to first file in archive. Returns a string with the name of current file, or null if not found.

Parameters

This function takes no parameters.

nextFile()

Advances file iterator to next file in archive. Returns a string with the name of current file, or null if not found.

Parameters

This function takes no parameters.

getFileName()

Returns name of current file in archive.

Parameters

This function takes no parameters.

extractFile()

Extracts the current file to the specified file.

Parameters

`destFile` - specifies the VFS path of the destination file.

close()

Closes archive file.

Parameters

This function takes no parameters.

Examples

```
var myZip = new Unzip();
myZip.open("zip/files.zip");
myZip.extractAll("files");
myZip.close();
```

To extract each file individually:

```
var myZip = new Unzip();
myZip.open("zip/files.zip");
var filename = myZip.firstFile();
while (filename != null)
{
    myZip.extractFile("files/" + filename);
    filename  = myZip.nextFile();
}
myZip.close();
```

Global Functions

MediaRich supports all the basic ECMAScript capabilities, including conditionals, variables, functions, and exception handling, as well as the proprietary image processing functions described here. For information on functions and objects not described in this guide, refer to the ECMAScript specification at `http://www.ecma.ch/ecma1/STAND/ECMA-262.htm` or to a JavaScript reference guide.

MediaRich also allows users to perform data queries and interact with ODBC-compliant databases through the Database, Cursor, and Stproc objects. For information on how to use these objects, go to `ftp://ftp.nombas.com/pub/isdkeval/se430/manual/c/html/TH_15927.htm`.

Working with Media Processing Functions

MediaScript is able to execute a number of media processing functions, or transforms, that you can use to generate almost any type of graphic.

Each function has a specific syntax and most include specific parameters that you can use to provide the information needed to execute the transform as desired. There are also generic parameters (non-executing parameters and multi-frame parameters) that apply to all transform commands.

Non-Executing Parameters

All transform commands can use two informational parameters:

`ParamInfo` - if specified, all the other parameters of the command are ignored. Instead a list of the legal parameters for that command are printed to the logfile (or screen).

`ParamCheck` - if specified, the specified function parameters are parsed and checked for legality and any resulting errors are returned, but the command does not execute.

Multi-Frame Parameters

Multi-frame files (such as GIF and TIFF) are a special case, because the files contain more than one frame. MediaScript supports a "frames" parameter that can be included with all transform commands for processing specific frames within a multi-frame file.

NOTE: *Multi-frame parameters can be used with the* `save()` *command, but not the* `load()` *command.*

`frames` - specifies a single frame, or a complex group of frames using a frame list. A frame list must be enclosed in quotes, and allows a comma separated list of individual frames or ranges. You can also specify a frame skip parameter to apply the relevant command to every *nth* frame.

NOTE: *The first frame in a multi-frame file is frame "1".*

Examples

```
image.load(name @ "clock.gif");
image.flip(axis @ "horizontal", frames @ "5");
image.save(name @ "flip5.gif");
```

This script flips the 5th frame of the file clock.gif.

```
image.load(name @ "clock.gif");
image.flip(axis @ "horizontal", frames @ "1,5,7-10");
image.save(name @ "multiflip.gif");
```

This script flips the 1st, 5th, 7th, 8th, 9th, and 10th frames of the file clock.gif.

```
image.load(name @ "clock.gif");
image.flip(axis @ "horizontal", frames @ "4-10-2");
image.save(name @ "flipskip.gif");
```

This script flips the 4th, 6th, 8th, and 10th frames (for example, every 2nd frame) of the file clock.gif.

MediaScript Global Functions

The MediaScript global functions are:

- CmykColor
- COMCreateObject()
- error()
- getPropertyValue()
- getScriptFileName()
- print()
- rgb()
- RgbColor Object
- version()

The global functions, CmykColor, RgbColor, and LoadAsRGB are defined in "Sys/color.ms".

CmykColor

This object is constructed from a 32-bit value. It splits the color into cyan, magenta, yellow, and black components.

Constructor

`CmykColor(value)` - returns an CmykColor object constructed from value.

Properties

`.cyan` - the cyan component (read or write). Valid values range from 0 to 255.

`.magenta` - the magenta component (read or write). Valid values range from 0 to 255.

`.yellow` - the yellow component (read or write). Valid values range from 0 to 255.

`.black` - the black component (read or write). Valid values range from 0 to 255.

Methods

`valueOf()` - converts the cyan, magenta, yellow, and black components back to a 32-bit value.

`toString()` - returns string representation of 32-bit value.

Functions

`CmykColorFromCMYK(cyan, magenta, yellow, black)` - constructs CmykColor from the components.

`CmykColorFromPct(cyan, magenta, yellow, black)` - constructs CmykColor from component percentages (0-1).

Examples

```
#include "Sys/color.ms"
        myColor = new CmykColor();
        myColor.cyan = 27;
        myColor.yellow = 122;
        myColor.magenta = 115;
        myColor.black = 55;
        media = new Media();
        media.makeCanvas(xs @ 100, ys @ 100, fillcolor @ myColor);
```

COMCreateObject()

COM objects can be created using a global function.

Parameters

`progId` - specifies a string containing the friendly progID of the COM object.

Examples

The following script uses the MediaGenClient COM object to call another MediaScript.

```
function main()
{
  var mgen = COMCreateObject("MediaGenClient.MGClient");
  mgen.ScriptName = "testtext.ms";
  mgen.SetParameter("args", 20);
  mgen.ExecuteScript();
  mgen.SaveBuffer(System.getNativePath("write:/comsave.jpg"));
  resp.setPath("write:/comsave.jpg");
}
```

NOTE: *This example is not generally recommended as it can cause deadlocks. It is provided for instructional purposes only.*

error()

This function has been deprecated. Use the standard JavaScript `try..catch..finally` and throw syntax instead.

getPropertyValue()

Returns a string with the value of the named property. If the named property does not exist, returns undefined. MediaRich includes two properties files that specify various system settings. The files are: `local.properties` and `global.properties`. Using the `getPropertyValue()` function, you can access these files from within a MediaScript.

NOTE: *Properties are controlled by the MediaRich system administrator using the Admin Center. See MediaRich Administrator's Guide for more information.*

Syntax

```
getPropertyValue(
       <"property name">
);
```

Parameters

Enter the property name in quotes. Property names consist of the filename in which the property exists (excluding the extension), a ".", and then the actual property name.

NOTE: *Property name information is case-sensitive.*

Example

To access the LogLevel property in the `local.properties` files:

```
getPropertyValue("local.LogLevel");
```

The `local.properties` file includes the following line:

```
LogLevel=error
```

So, `getPropertyValue()` returns "error".

getScriptFileName()

Returns the current filename for this running script.

Syntax

```
getScriptFileName();
```

print()

Depending on the platform, prints the specified string to the command prompt (MS Windows) or to MediaScript log (Mac OS).

Syntax

```
print (<"string">);
```

rgb()

Converts the three supplied RGB color values into a 24-bit value (0 - 16,777,215) that is suitable for many `Media()` graphic operation arguments.

Syntax

```
rgb (<red>,<green>,<blue>);
```

RgbColor Object

This object is constructed from a 24-bit value. It splits the color into red, green, and blue components.

Constructor

`RgbColor(value)` - returns an RgbColor object constructed from value.

Properties

`.red` - the red component (read or write). Valid values range from 0 to 255.

`.green` - the green component (read or write). Valid values range from 0 to 255.

`.blue` - the blue component (read or write). Valid values range from 0 to 255.

Methods

`valueOf()` - converts the red, green, and blue components back to a 24-bit value.

`toString()` - returns string representation of 24-bit value.

Functions

`RgbColorFromRGB(red, green, blue)` - constructs RgbColor from the components.

`RgbColorFromPct(red, green, blue)` - constructs RgbColor from component percentages (0-1).

Examples

```
#include "Sys/color.ms"
        myColor = new RgbColor(0x1133aa);
        print (myColor.red, myColor.green, myColor.blue);
        myColor = new RgbColor();
        myColor.red = 27;
        myColor.green = 59;
        myColor.blue = 255;
        media = new Media();
        media.makeCanvas(xs @ 100, ys @ 100, fillcolor @ myColor);
```

version()

Returns a string that is the current version of MediaScript.

Syntax

```
version();
```

Appendix A
MediaRich Metadata Support

MediaRich provides support for the most popular metadata formats: IPTC, Exif, and XMP. MediaRich fully supports loading, saving and merging IPTC, Exif, and XMP metadata for JPEG, TIFF, and Photoshop files. MediaRich also supports loading XMP metadata from the following file formats: Illustrator, EPS, GIF, PDF, and PNG. This metadata is available to the script as a metadata XML document. Detailed schemas are provided for the Exif and IPTC documents constructed by MediaRich. The XMP metadata document conforms to the schema defined by Adobe.

The MediaRich for SharePoint™ product currently supports only IPTC and Exif metadata for JPEG, TIFF, and Photoshop files.

Appendix Summary

Low-Level Metadata Interface

Two MediaScript objects provide support for metadata: The Media object and the _MR_Metadata object.

The Media object provides support for loading and saving metadata along with the image contents.

The _MR_Metadata object provides support for loading and merging just the metadata contained within the files, without loading the image data. This allows the scripter to modify the metadata within compressed files without decompressing and recompressing the image data.

The Media Object

The load command of the Media object will load and attach Exif, IPTC, and XMP metadata if the `loadMetadata` parameter to the load command is specified as true. For example:

```
var image = new Media();
image.load(name @ "myimage.jpg", loadMetadata @ true);
```

This constructs an XML document for any Exif, IPTC, or XMP metadata contained in the file and attach it to the Media object. This metadata can be accessed by the scripter using the `getMetadata` command of the Media object.

```
var metaDoc = image.getMetadata("IPTC");
```

This metadata document can be processed and edited. To modify the metadata attached to the image, use the `setMetadata` method:

```
image.setMetadata("Exif", myExifData);
```

Finally, use the save method to save any metadata attached to the document unless the `SaveMetadata` parameter is set to false.

```
image.save(type @ "jpeg");
```

The metadata names for use with the `getMetadata` and `setMetadata` methods are "Exif", "IPTC", and "XMP" for the Exif, IPTC, and XMP metadata documents, respectively. These names are case-sensitive.

NOTE *A save() function will embed a color profile for a color profile-supporting format without explicitly setting saveMetadata to "true" only when CMYK data is present.*

The _MR_Metadata Object

The _MR_Metadata object supports extracting metadata from supported file formats without loading the image data. It also supports merging new metadata into existing files without the need to interpret or decompress the image data. The _MR_Metadata object has two methods: save, and load. In addition, the _MR_Metadata object can be used as the MediaScript response object allowing the script to stream back a file with modified metadata.

The _MR_Metadata constructor takes a file name and an optional file type.

```
var metaObj = new _MR_Metadata("myimage.jpg", "jpeg");
```

If the file has a valid extension, the file type may be omitted.

The _MR_Metadata save command provides a single object as a parameter, in a manner similar to the Media object save command. This parameter may be specified as an object, or using the ampersand ("@") notation.

The parameters for the save command object are exif, iptc, xmp, and name. The file type of the result is always the same as the file type of the original image.

```
metaObj.save( exif @ myExifDoc, iptc @ null, xmp @ null,
name @ "newFile");
```

or

```
var saveObj = new Object();
saveObj.iptc = null;
saveObj.xmp = null;
saveObj.exif = myExifDoc;
saveObj.name = "newFile";
metaObj.save(saveObj);
```

If any of the exif, iptc, or xmp parameters are omitted, existing metadata of that type in the file will be transferred to the output file. If any of the exif, iptc, or xmp parameters are specified as null, existing metadata of that type will be omitted in the output. Otherwise, IPTC and XMP data will be replaced with the specified data, and writable Exif tags will be replaced. The Exif camera data tags are never replaced.

High-Level Support for Exif and IPTC

Two MediaScript objects are provided to simplify the tasks of getting and setting individual metadata items. These objects are provided to support IPTC and Exif metadata. Each of these objects has a similar format, providing set<Tag> methods and get<Tag> methods which set and get individual metadata fields, respectively. <Tag> represents the name of the metadata tag to set or get.

The following section describes the general structure and common methods for both the IPTCMetadata object and the ExifMetadata object. This is followed by descriptions of the set<Tag> and get<Tag> methods for the IPTCMetadata and ExifMetadata objects.

Common Metadata Methods

The IPTCMetadata and ExifMetadata objects have several common methods allowing the scripter to create documents, specify metadata for existing documents, extract a string representation of the XML document, and validate the document. These operations are summarized in the following table.

Method	Description
`IPTCMetadata(validate)`	Constructs a blank IPTC metadata document. If validate is true, the document is automatically validated in set<tag> methods.
`ExifMetadata(validate)`	Constructs a blank Exif metadata document.
`loadFromFile(filename)`	Load metadata object with data from the specified file.
`loadFromMedia(media)`	Load metadata object with data from the specified media.
`loadFromXML(xmlString)`	Load metadata object with data from the specified XML string.
`blankDocument()`	Load metadata object with a valid blank document.
`validate()`	Validates the document to the appropriate schema.
`isEmpty()`	Returns true if the document is empty.

NOTE: *Each metadata constructor constructs a blank metadata document of the appropriate type. This document is not empty. It is a valid XML document for the appropriate metadata schema. However, loadFromFile and loadFromMedia will leave the document in an empty state if the file contains no metadata of the appropriate type.*

```
var metadata = new IPTCMetadata();
var empty = metadata.isEmpty(); // returns false
metadata.loadFromFile("img.jpg");
if (!metadata.isEmpty())
{
 // do something with metadata
}
else
```

```
{
  // file did not contain metadata.
}
```

Alternatively, you can simply use the get<Tag> methods, which will return null if the document is empty.

IPTCMetadataObject

The following methods may be used to get and set metadata values for IPTC metadata. Please refer the schema (`IPTC.xsd`) in the `//Shared/Originals/Sys` folder for the required format for each of the IPTC fields. Note that only a brief description is provided here. For a complete description of each IPTC metadata field, please consult the IPTC news-photo metadata specification available at `http://www.iptc.org` under the title "Digital Newsphoto Parameter Record".

In the following table, the notation (string ...) indicates that multiple values may be specified as arguments to the method.

Method	Description
`getVersion()`	Returns the version field.
`setVersion(string)`	Sets the version field.
`getObjectTypeReference()`	Returns the object type reference field.
`setObjectTypeReference(string)`	Sets the object type reference field.
`getObjectAttributeReference()`	Returns an array of attribute references.
`setObjectAttributeReference (string, ...)`	Sets attribute references.
`addObjectAttributeReference (string, ...)`	Adds an attribute references to the list.
`setObjectAttributeReferenceArray (array)`	Sets a group of attribute references from an Array.
`getObjectName()`	Returns the object name.
`setObjectName(string)`	Sets the object name.
`getEditStatus()`	Returns the edit status.
`setEditStatus(string)`	Sets the edit status.
`getEditorialUpdate()`	Returns the editorial update code.
`setEditorialUpdate(string)`	Sets the editorial update code.
`getUrgency()`	Returns the urgency code.
`setUrgency(string)`	Sets the urgency code.
`getSubjectReference()`	Returns an array of subject references.
`setSubjectReference(string, ...)`	Sets subject references.

Method	Description
`addSubjectReference(string, ...)`	Adds subject references to the list.
`setSubjectReferenceArray(array)`	Sets a group of subject references from an Array.
`getCategory()`	Returns the category code.
`setCategory(string)`	Sets the category code.
`getSupplementalCategory()`	Returns the supplemental category array.
`setSupplementalCategory(string, ...)`	Sets supplemental categories.
`addSupplementalCategory(string, ...)`	Adds a value to the supplemental category list.
`setSupplementalCategoryArray(array)`	Sets a group of supplemental categories as an Array.
`getFixtureIdentifier()`	Returns the fixture identifier code.
`setFixtureIdentifier(string)`	Sets the fixture identifier code.
`getKeywords()`	Returns an array of keywords.
`setKeywords(string, ...)`	Sets keywords.
`addKeywords(string, ...)`	Adds keywords to the list
`setKeywordsArray(array)`	Sets keywords from an Array.
`getContentLocation()`	Gets an array of content location objects. Each object has two properties: ContentLocationName and ContentLocationCode.
`getContentLocationName(which)`	Returns the ContentLocationName subfield of the ContentLocation tag indexed by which.
`getContentLocationCode(which)`	Returns the ContentLocationCode subfield of the ContentLocation tag indexed by which.
`setContentLocation(name, code)`	Sets the ContentLocation tag to the specified name and code.
`addContentLocation(name, code)`	Adds the ContentLocation specified by name and code to the ContentLocation list.
`getReleaseDate()`	Returns the ReleaseDate and ReleaseTime tags as a MediaScript Date object.
`setReleaseDate(date)`	Sets the ReleaseDate and ReleaseTime tags from a MediaScript Date object.
`setReleaseTime(string)`	Sets only the ReleaseTime field.
`getExpirationDate()`	Returns the ExpirationDate and ExpirationTime fields as a MediaScript Date object.
`setExpirationDate(date)`	Sets the ExpirationDate and ExpirationTime fields from a MediaScript Date object.
`setExpirationTime(string)`	Sets only the ExpirationTime field.
`getSpecialInstructions()`	Returns the SpecialInstructions field.

Method	Description
`setSpecialInstructions(string)`	Sets the SpecialInstructions field.
`getActionAdvised()`	Returns the ActionAdvised field.
`setActionAdvised(string)`	Sets the ActionAdvised field.
`getReference()`	Returns an array of Reference object for the Reference field. Each reference object has a ReferenceService, ReferenceDate, and ReferenceNumber property.
`getReferenceService(which)`	Returns the ReferenceService property of the Reference element indexed by which.
`getReferenceDate(which)`	Returns the ReferenceDate property of the Reference element indexed by which as a MediaScript Date object.
`getReferenceNumber`	Returns the ReferenceNumber property of the Reference element indexed by which.
`setReference(service, date, number)`	Sets the Reference element to the reference specified by service, date and number. Date must be a MediaScript Date object.
`addReference(service, date, number)`	Adds a Reference element to the list using the specified service, date and number. Date must be a MediaScript Date object.
`getDateCreated()`	Returns the DateCreated and TimeCreated fields as a MediaScript Date object.
`setDateCreated(date)`	Sets the DateCreated and TimeCreated fields from a MediaScript Date object.
`setTimeCreated(string)`	Sets the TimeCreated field.
`getDigitalCreationDate()`	Returns the DigitalCreationDate and DigitalCreationTime fields as a MediaScript Date object.
`setDigitalCreationDate(date)`	Sets the DigitalCreationDate and DigitalCreationTime fields from a MediaScript Date object.
`setDigitalCreationTime(string)`	Sets the DigitalCreationTime field.
`getOriginatingProgram()`	Gets the OriginatingProgram field.
`setOriginatingProgram(string)`	Sets the OriginatingProgram field.
`getProgramVersion()`	Gets the ProgramVersion field.
`setProgramVersion(string)`	Sets the ProgramVersion field.
`getObjectCycle()`	Gets the ObjectCycle field.
`setObjectCycle(string)`	Sets the ObjectCycle field.

Method	Description
`getByLine()`	Returns an array of ByLine objects, each of which contains a ByLineWriter and a ByLineTitle property.
`getByLineWriter(which)`	Returns the ByLineWriter property of the ByLine element specified by which.
`getByLineTitle(which)`	Returns the ByLineTitle property of the ByLine element specified by which.
`setByLine(writer, title)`	Sets the ByLine element to the specified writer and title.
`addByLine(write, title)`	Adds an element to ByLine for the given writer and title.
`getCity()`	Returns the City element.
`setCity(string)`	Sets the City element.
`getSublocation()`	Returns the Sublocation.
`setSublocation(string)`	Sets the Sublocation.
`getState()`	Returns the State (Province).
`setState(string)`	Sets the State (Province).
`getCountryCode()`	Returns the CountryCode.
`setCountryCode(string)`	Sets the CountryCode.
`getCountryName()`	Returns the CountryName.
`setCountryName(string)`	Sets the CountryName.
`getOriginalTransmissionReference()`	Returns the OriginalTransmissionReference.
`setOriginalTransmissionReference (string)`	Sets the OriginalTransmissionReference.
`getHeadline()`	Returns the Headline.
`setHeadline(string)`	Sets the Headline.
`getCredit()`	Returns the Credit.
`setCredit(string)`	Sets the Credit field.
`getSource()`	Returns the Source field.
`setSource(string)`	Sets the Source field.
`getCopyrightNotice()`	Returns the Copyright field.
`setCopyrightNotice(string)`	Sets the Copyright field.
`getContact()`	Returns an array of Contact elements.
`setContact(string, ...)`	Sets Contact elements.
`addContact(string, ...)`	Adds Contact elements.
`setContactArray(array)`	Sets Contact element from an Array.

Method	Description
`getCaption()`	Returns the Caption element.
`setCaption(string)`	Sets the Caption element.
`getWriter()`	Returns an array of writer elements.
`setWriter(string, ...)`	Sets Writer elements.
`addWriter(string, ...)`	Adds Writer elements.
`setWriterArray(array)`	Sets Writer elements from an array.
`getImageType()`	Returns the ImageType.
`setImageType(string)`	Sets the ImageType.
`getImageOrientation()`	Returns the ImageOrientation.
`setImageOrientation(string)`	Sets the ImageOrientation.
`getLanguageIdentifier()`	Returns the LanguageIdentifier.
`setLanguageIdentifier(string)`	Sets the LanguageIdentifier.

The Exif Metadata Object

The following methods may be used to get and set metadata values for Exif metadata. Please refer the schema (`Exif.xsd`) in the `//Shared/Originals/Sys` folder for the required format for each of the Exif fields. Note that only a brief description is provided here. For a complete description of each Exif metadata field, please consult the Exif metadata specification available at `http://www.exif.org`.

NOTE: *Where possible, the values listed in the following tables are converted into string valued representations as defined in the exif specification.*

IFDO Methods

Method	Description
`getImageDescription()`	Returns the image description.
`setImageDescription(string)`	Sets the image description.
`getOrientation()`	Returns the image orientation.
`setOrientation(string)`	Set the image orientation.
`getSoftware()`	Returns the software description.
`setSoftware(string)`	Sets the software description.
`getArtist()`	Returns the artist.
`setArtist(string)`	Sets the artist.
`getDateTime()`	Returns the DateTime field as a MediaScript Date object.

Method	Description
`setDateTime(date)`	Sets the DateTime field from a MediaScript Date object.
`getPhotographerCopyright()`	Returns the photographer copyright.
`setPhotographerCopyright(string)`	Sets the photographer copyright.
`getEditorCopyright()`	Returns the editor copyright.
`setEditorCopyright(string)`	Sets the editor copyright.
`getMake()`	Returns the camera make.
`getModel()`	Returns the camera model.
`getImageWidth()`	Returns the image width.
`getImageLength()`	Returns the image height.
`getBitsPerSample()`	Returns the number of bits per sample.
`getCompression()`	Returns the compression type.
`getPhotometricInterpretation()`	Returns the photometric interpretation.
`getPlanarConfiguration()`	Returns the planar configuration.
`getYCbCrSubSampling()`	Returns the YCbCr sub-sampling.
`getYCbCrPositioning()`	Returns the YCbCr positioning.
`getXResolution()`	Returns the horizontal resolution.
`getYResolution()`	Returns the vertical resolution.
`getResolutionUnit()`	Returns the resolution unit.
`getWhitePoint()`	Returns the white point.
`getPrimaryChromaticities()`	Returns the primary chromaticities.
`getYCbCrCoefficients()`	Returns the YCbCr coefficients.
`getReferenceBlackWhite()`	Returns the ReferenceBlackWhite value.

IFDExif Methods

Method	Description
`getVersion()`	Returns the Exif version.
`setVersion(string)`	Sets the Exif version (default = 2.1)
`getFlashPixVersion()`	Returns the flashpix version.
`setFlashPixVersion(string)`	Sets the flashpix version
`getUserComment()`	Returns the user comment.
`setUserComment(string)`	Sets the user comment.
`getColorspace()`	Returns the colorspace.

Method	Description
`getPixelXDimension()`	Returns the width.
`getPixelYDimension()`	Returns the height.
`getComponentsConfiguration()`	Returns the ComponentsConfiguration value.
`getCompressedBitsPerPixel()`	Returns the approx. number of compressed bits per pixel.
`getRelatedSoundFile()`	Returns the name of a related sound file.
`getDateTimeOriginal()`	Returns a MediaScript Date object for the original date/time.
`getDateTimeDigitized()`	Returns a MediaScript Date object for the digitized date/time.
`getSubSecTime()`	Returns the sub-second time offset.
`getSubSecTimeOriginal()`	Returns the sub-second time offset for the original.
`getSubSecTimeDigitized()`	Returns the digitized sub-second time offset.
`getExposureTime()`	Returns the exposure time.
`getShutterSpeedValue()`	Returns the shutter speed in seconds.
`getApertureValue()`	Returns the aperture value as an F-number.
`getBrightnessValue()`	Returns the brightness value.
`getExposureBiasValue()`	Returns the exposure bias value.
`getMaxApertureValue()`	Returns the maximum aperture value as an F-number.
`getSubjectDistance()`	Returns the subject distance in meters.
`getMeteringMode()`	Returns the metering mode.
`getLightSource()`	Returns the light source.
`getFlash()`	Returns true if flash was used.
`getFocalLength()`	Returns the focal length.
`getFNumber()`	Returns the F-number.
`getExposureProgram()`	Returns the exposure program.
`getSpectralSensitivity()`	Returns the spectral sensitivity.
`getISOSpeedRatings()`	Returns the ISO film speed.
`getOECF()`	Returns the OECF value.
`getFlashEnergy()`	Returns the flash energy.
`getSpatialFrequencyResponse()`	Returns the spatial frequency response.
`getFocalPlaneXResolution()`	Returns the focal-plane horizontal resolution.
`getFocalPlaneYResolution()`	Returns the focal-plane vertical resolution.

Method	Description
`getFocalPlaneResolutionUnit()`	Returns the focal-plane resolution unit.
`getSubjectLocation()`	Returns the subject location.
`getExposureIndex()`	Returns the exposure index.
`getSensingMethod()`	Returns the sensing method.
`getFileSource()`	Returns the file source.
`getSceneType()`	Returns the scene type.
`getCFAPattern()`	Returns the CFA pattern.

Appendix B
MediaRich Color Management

MediaScript uses the ICC (International Color Consortium) methodology for color management, where the color characteristics of each color reproduction device to be used is stored in a color profile.

This appendix explains the basics of color management and how to use MediaRich and the MediaScript scripting language to insure accurate RGB and CMYK color conversions.

Appendix Summary

Color Management Overview

MediaScript uses the ICC (International Color Consortium) methodology for color management, where the color characteristics of each color reproduction device to be used is stored in a color profile. Converting an image from one device to another involves setting up a color transformation between the source profile and the destination profile. This transformation is then used to convert each pixel in the image from the source device to the destination device. Detailed information regarding ICC color management can be found at the ICC web site.

Colorspaces

Different color reproduction devices use different types of color representation (colorspaces) for color reproduction. Most color reproduction devices fall into three basic categories. Each category is represented by a different image colorspace.

Grayscale colorspace devices use a single intensity corresponding to the darkness/lightness desired. In MediaScript, a grayscale value of 0 represents black, while a grayscale level of 255 represents white.

RGB colorspace devices (typically monitors) use red, green, and blue intensities which combine to form the color. The value combined additively; when all three components are 0, the color is the darkest the device can represent; and when all three components are 255, the color is the brightest.

CMYK colorspace devices (typically printers) use the intensity of the cyan, magenta, yellow, and black inks to represent the color. When no ink is used, (all components 0), the color is the color of the paper. When the maximum intensities are used, the color is at its darkest. Note that most printers cannot print using the maximum value of all four channels.

Converting colors between these different types of devices fundamentally changes the nature of the image data. For example, the color white on an RGB device is typically represented by setting all three channels to 255. However, on a CMYK device, white is typically represented by setting all four channels to 0. It is therefore important to know which colorspace an image is represented in.

Color Gamut

Different devices may not be able to reproduce the same range of colors. The range of colors that a given device can reproduce is called its “gamut”. RGB devices typically have a significantly different gamut than CMYK devices. Therefore, converting an image from an RGB colorspace to a CMYK colorspace or vice-versa typically involves a loss of information as different colors in the source colorspace may map to the same color in the destination colorspace. For this reason, it is advisable to keep the number of color conversions to a minimum.

White Point Mapping

The color considered to be “white” on each device may not be the same. What is considered “white” for CMYK devices depends on the paper being used, while “white” on a monitor depends on the maximum intensities of the red, green, and blue channels. Typically, monitors have a “white” that is noticeably bluer than paper.

However, when an image displayed on the screen is printed, one does not typically expect the image to be printed on a blue background. Instead, the “white” of the monitor is mapped to the “white” of the printer, such that areas displayed as “white” on the monitor are printed with no ink. This is termed white point mapping.

Color Profiles

Color profiles for a given device can typically be obtained from the device manufacturer, or may come installed on the operating system. Additionally, software is available that can be used to characterize a device and create a color profile. The TIFF, JPEG, EPS, and Photoshop image formats have the ability to store the color profile that describes the image contents along with the image. This is termed an “embedded profile”.

MediaScript reads embedded profiles from these image formats and will preferentially use these profiles as the source for color transformations. Additionally, MediaScript has the ability to save the embedded profile along with the image data.

Rendering Intent

Because of differences in color gamut and white point between differing devices, building a transformation between devices requires the user to specify how to handle gamut and white point mapping. This is specified as the rendering intent. The ICC provides four different intents:

- Perceptual
- Relative Colorimetric
- Absolute Colorimetric
- Saturation

Perceptual

This intent specifies that the white points of the differing device be mapped, and that color differences for out of gamut colors be preserved. This implies that some compression of colors that lie inside both gamut be performed to make room for different out-of-gamut colors. This is usually the best rendering intent for complex images such as photographic images.

Relative Colorimetric

This intent specifies that colors that lie inside the common color gamut of both devices be reproduced exactly. Colors that are outside of the destination devices color gamut are mapped to the gamut boundary. This means that many out of gamut colors may be

mapped to the same color on the destination device. This intent is useful when it is known that the colors in the image mainly lie inside the gamut of both the source and the destination device.

Absolute Colorimetric

This intent is used when you want the colors on the destination device to be colorimetrically equal to that on the source device. No white point mapping is done. This intent is rarely used for image processing.

Saturation

For certain types of images (notably business graphics), it is more important to preserve the saturation of the color than the color itself. The saturation intent is provided to handle this case.

MediaScript Color Management Functions

The color management functions provided by MediaScript can be divided into two categories:

- image conversion
- profile management.

The image conversion functions are all members of the Media object, while the profile management functions are provided by the IccProfile object. A brief overview of these functions is provided here.

Image Conversion Methods

The image conversion methods are all member functions of the Media object.

- The `colorCorrect` method is the primary method used for converting between a source device, represented by the source profile, and a destination device represented by the destination profile.
- The `colorToImage` method provides a mechanism for converting a single color from a specified source to the image colorspace.
- The `colorFromImage` method converts a single color from the image colorspace to a specified destination.
- The `embeddedProfile` method tests an image to see if it has an embedded profile.
- The `getEmbeddedProfileName` method returns the name of the embedded profile.
- The `saveEmbeddedProfile` method will save the embedded profile to a file.
- The `setSourceProfile` method sets the embedded profile for an image to the specified profile.

For specific details of these functions see “Media Object” on page 43.

Profile Management methods

With the exception of the load and save methods of the Media object, the profile management methods are contained in the IccProfile link object.

Media Object

The `load()` method of the Media object automatically loads any embedded source profiles for supported file formats.

The save method can optionally embed the color profile associated with an image along with the image data for supported file formats. By default, embedded profiles are always saved for CMYK images, but not for RGB images. This default behavior is controlled by the `ColorManager.DefaultEmbedProfile` property in the `global.properties` file.

IccProfile Object

The IccProfile object constructor will construct an IccProfile object given either a Media object that has an embedded profile or a path to a profile on disk. An IccProfile object may be used as a source or destination profile for all color management methods that take profiles as arguments, including:

- getPath()
- getName()
- getClass()
- getColospace()
- getConnectionspace()
- list()

getPath()

The getPath method returns the path to the profile on disk or the empty string if the object was constructed from an image.

getName()

The getName method returns the profile name.

getClass()

The getClass method returns the profiles class. The class generally describes the type of the device for which the profile is intended.

getColospace()

The getColorspace method returns the colorspace of the device characterized by the profile. This is the colorspace of the source image if the profile is used as the source, and the colorspace of the destination image if the profile is used as the destination.

getConnectionspace()

The getConnectionspace method returns the common colorspace used to construct transforms using this profile.

list()

The list method is a static method that returns an array of profile names that match specified class and colorspace criteria. This method may be used for example to return a list of all Monitor profiles that have an RGB colorspace.

Specifying Profiles

The methods `colorCorrect`, `colorFromImage`, `colorToImage`, and `setSourceProfile` all take a SourceProfile argument, a DestProfile argument, or both.

There are three methods for specifying these profiles:

- Specify the path to a profile on disk. By default this path is relative to the "color:" virtual file system which includes both the `Originals/Profiles` directory in the Shared files folder and the system color profile directory;
- Use an IccProfile object. These object may be constructed either from a path to a profile on disk, or from an image with an embedded profile;
- Use the default profiles by specifying the string "rgb" for the default RGB colorspace profile, or "cmyk" for the default CMYK colorspace profile. These default profiles are defined in the `global.properties` file.

Accuracy and Reversibility of Color Conversions

The quality of the color reproduction of an image converted from one colorspace to another depends on the following factors:

- The overlap between the gamut of the source device and the gamut of the destination device;
- The quality of the color profile used (i.e., the number of samples used to generate the conversion transform);
- The accuracy of the color profile;
- The rendering intent selected.

Devices with similar gamuts will produce the smallest color distortion. Typically, converting from one monitor's colorspace to another monitor's colorspace results in no noticeable loss of color quality. However, converting from an RGB colorspace to a CMYK colorspace will typically result in changes in the color from the original. This loss is caused by the fact that monitors typically have larger gamuts than printers. For photographic images, this loss is typically small since most naturally occurring colors can be produced equally well on monitors and printers. For computer-generated images, the loss is typically larger since there are many colors representable on a monitor that cannot be achieved with ink on paper.

Printer profiles are usually constructed with a large number of sample points for converting colors to the printers' colorspace. However, to make the profiles reasonably small, far fewer sample points are usually provided for converting from the printers' colorspace. Therefore, color conversions using printer profiles as the source colorspace should be avoided whenever possible.

Because of the inherent inaccuracy in the color conversion process, converting from one colorspace to another and back should also be avoided where possible. Images for processing should be converted to a common colorspace, processed, and converted to the destination. Choice of the common colorspace depends on the colorspace of the images involved, the type of images involved, the quality of the color profiles, the destination colorspace, and the operations to be performed.

Common Color Management Questions

How do I color correct RGB images that do not have embedded profiles?

Specify a source profile to the `colorCorrect` function. This source profile will only be used for images that do not have an embedded source profile. If you specify the string "rgb" as the source profile for colorCorrect, the default RGB profile specified in the `global.properties` file will be used for any image that lacks a source profile.

```
image.colorCorrect(SourceProfile @ "rgb",
DestProfile @ myPrinterProfile, Intent @ "RelativeColorimetric");
```

How do I composite an RGB image on to an CMYK image?

You must first convert the RGB image to the CMYK colorspace of the image. The simplest method for performing this conversion is to construct an IccProfile object from the CMYK image. This IccProfile object may then be used as the destination profile for the conversion.

For example, suppose that the CMYK image is contained in a Media object named cmykImage, and the RGB image is contained in a Media object named rgbImage. The following script will perform the composite:

```
cmykProfile = new IccProfile(cmykImage);
rgbImage.colorCorrect(SourceProfile @ "rgb",
DestProfile @ cmykProfile, Intent @ "Perceptual");
cmykImage.composite(source @ rgbImage);
```

NOTE: *The new `IccProfile(cmykImage)` method will throw an exception if cmykImage does not contain an embedded profile.*

When I try to construct an IccProfile object, I get the "Not a function type" error. What's wrong?

The IccProfile object is a MediaScript link object. To use the IccProfile object when MediaRich is installed on a Windows server, type the following line at the beginning of your script.

```
#link <IccProfile.dll>
```

What's the best method for color correcting a Grayscale image?

To maintain compatibility with previous versions of MediaRich, Grayscale images may be treated in the same manner as RGB images. However, this assumes that the default RGB colorspace uses equal amounts of red, green, and blue to represent gray, and that the transform from Grayscale to RGB is linear.

If a profile is available for your Grayscale images, better results will be obtained by using the Grayscale profile to color correct the Grayscale image to RGB before processing it as an RGB image.

How do I treat all images as RGB images?

If you are not concerned about precise color conversions, the loadAsRGB method has been provided as an add-on to the media object in the file: `Scripts/Sys/media.ms`.

This method loads an image, and if the image colorspace is not RGB, it converts the image to RGB using the embedded profile, default source, and destination profiles defined in the global.properties file. If the image has an embedded profile, it is used as the source. Here is an example of how to use the loadAsRGB method:

```
#include "Sys/media.ms"
image = new Media();
image.loadAsRGB(name @ "myImage.tif");
```

Now, irrespective of the colorspace of the image stored in `myImage.tif`, the image is an RGB image.

Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z