MediaRich® SERVER

MediaRich CORE 6.2

Programmer's Guide

Published: September 2020

equilibrium®

Contents

CHAPTER I

MediaRich CORE Programming

The MediaRich CORE Platform is designed specifically to make it easy to create images from flexible and efficient image templates. Video, imaging, documents, and Adobe .pdf, .eps, and .ai files are now available for rendering by the MediaRich CORE.

Important: A/V Core 2.0 is only available on Windows Operating Systems at this time.

The platform consists of several components:

MediaRich CORE

The MediaRich CORE is a multi-threaded application server which accepts incoming requests and either returns a previously cached response or executes the appropriate MediaScript to handle the request.

MediaScript

MediaScript is an interpreted scripting language based on the ECMAScript (commonly known as JavaScript) language specification. MediaScript extends ECMAScript with several new objects that provide media processing and general application server functionality. For more information, see "Using MediaScript" on page 36.

Client APIs

The client APIs allow end users to generate requests and process responses. MediaRich includes support for standard HTTP URLs, Web Services, .NET, Java and COM. For more information, see "MediaRich Client APIs" on page 8.

MediaRich uses a simple stateless protocol for forming image processing requests and streaming back the resulting image data.

The life-cycle of a MediaRich request follows this sequence of events:

1. *Request Creation.* Client creates a request to apply a specific set of parameters to original image. A MediaRich request consists of several elements: a file path, the name of a function to

execute (zoom, scale, etc.), arguments to that function, and other user- and system-defined parameters. The most common type of a MediaRich request is an HTTP URL which is usually embedded in HTML `<img src="…">` tag.

2. *Request Transmission.* After a request is created, it is sent to MediaRich. In most cases it is a HTTP URL, and web browser sends the request. Alternatively, applications can send requests directly to the Media Generator via Web Services.
3. *Cache Lookup.* When Media Generator receives a request, it first checks its internal cache for a previously generated and cached image. If one is found, it is returned as the response and the function execution (step 4) is skipped.
4. *MediaScript Execution.* The requested MediaScript function is executed using the arguments and parameters supplied in the request. The function sets the response object (usually an image) and whether the response should be cached or streamed directly to the client. When the script terminates, the response is sent to the client.
5. *Response Handling.* Finally, the client (usually web browser) processes the response. A response can be either a path (in the case of a cached or file path response) or the actual binary data (in the case of a streamed response).

CHAPTER 2

MediaRich Client APIs

The MediaRich platform provides five client APIs: HTTP, Web Service, Java, .NET, and COM. All five interfaces allow clients to create requests and send them to a MediaRich CORE. The MediaRich CORE will, in most cases, respond by generating an image based on the properties in the request and returning it to the calling client.

Chapter summary

API Overview

All five client APIs provide the same basic functionality, but are intended for use in different environments. HTTP URLs are usually embedded in HTML pages and can be generated dynamically by application servers or client-side scripts. When end users direct their web browsers to a page containing a MediaRich URL, the browser sends the URL to the MediaRich Web Server and then displays the returned image or other asset.

Client applications can use all of these APIs to communicate with a MediaRich server.

HTTP API

The HTTP API is the simplest of the APIs. It can be used by any application that can send requests to a HTTP server. The HTTP API is well suited to Web applications that use AJAX to talk to a server.

For detailed usage information, see "HTTP API" on page 12.

Web service API

The Web Service API is also fairly simple, and is used primarily by applications designed to integrate a number of Web services into a single application.

.NET and Java client APIs

The .NET and Java client APIs are the most powerful APIs. They allow programs written in Microsoft's .NET and Sun's Java development environments to communicate with MediaRich. These APIs can do everything that the HTTP and Web Service APIs can do, and a lot more.

COM client API

The COM client API allows programs and scripts that can utilize COM to communicate with a MediaRich server. The COM API is more powerful than the HTTP or Web Service APIs, but is more limited in functionality than the .NET and Java APIs. The COM API should be used by applications that do not have access to the .NET framework.

The Web Service, .NET and COM APIs are only applicable to writing client applications that will run on the Microsoft Windows platform. The HTTP and Java APIs can be used to write client applications that will run on any platform. All of the client APIs can make requests to any MediaRich CORE regardless of the platform on which it is running.

API Selection

To determine which MediaRich client API is the best fit, consider the nature of the client application you are writing and the development environment you will use to write that application.

Using the HTTP API

You can use he HTTP API in almost all situations. This is true because nearly every development language and environment provides a way to submit HTTP requests to a web server. MediaRich is a Web server, so any application that can submit HTTP requests can communicate with MediaRich using this protocol.

The HTTP protocol is less capable than some of the other protocols, but it has significant advantages if it fits your needs. Firstly, it is the simplest protocol, and therefore the easiest to understand and use. Secondly, you can always use a web browser during development to check your MediaRich scripts outside of your application environment.

Development application and environment

Reviewing the following questions will help you to choose the right API for your application. Use this process to identify the characteristics of your application and environment and point you to the best fit API for your needs.

Is your application web browser-based?

Your answer to this question will be “yes” if you are generating web pages that will contain media assets delivered by MediaRich. You might also answer “yes” here if you are writing an interactive web application, using DHTML, JavaScript and/or Ajax. If you do answer “yes” to this question, then the HTTP API is probably what you’ll want to use.

If you are writing a web application and you will access MediaRich from the web server rather than from code running on the client, the nature of the environment in which your server-side code will run should dictate the API to use. Review the other questions to determine which API to use in this case.

Is your application written in a Visual Studio .NET language or in Java?

If so, you’ll want to use the respective MediaRich API (.Net or Java). This will most likely be the case if you are writing a stand-alone client application. It will also be true if you are accessing MediaRich from server-side logic in a Web application, and you are writing that logic in ASP .NET or JSP.

Is your application designed to access web services using the web Services framework?

If so, you might want to use the Web Services API instead of one of the others. If the answers to prior questions suggest another API, the choice will be yours as to which API to use.

Is your application written in a “legacy” development environment on the Windows platform?

In other words, are you writing a Windows application that does not have access to the .NET framework?

The answer is “yes” if you are developing in an older version of Visual Studio, or if you are writing older style Visual Basic or Visual Java applications or scripts. In this case, you will want to use either the HTTP or COM API.

Are you writing code that must be cross-platform?

If so, you’ll want to use either the HTTP or Java APIs. The other APIs are applicable only to the Microsoft Windows environment.

Do you need to send file data, such as image, movie, or script files, from your client to the MediaRich server?

If so, you will need to use the .NET, Java, or COM API. The HTTP and Web Services APIs do not allow sending files or other large blocks of data to the server.

MediaRich Requests

A complete MediaRich request specifies the following:

- The MediaRich CORE that will handle the request
- The MediaScript script file that contains the function to be executed
- The name of the MediaScript function to be executed in that file
- A set of arguments and parameters to be passed to that function

The kind of result returned to the caller by the request is determined by the function that is executed, and can be any block of data along with a MIME type that indicates the type of that data. The type of result most often returned by MediaRich requests is an image file (such as a JPEG or GIF file). Other common result types are plain text, an HTML page, a XML document, or a video or audio file.

When the requested function is executed on the server, that execution can produce desirable side effects in addition to producing the result that is returned to the caller. For example, the function can write information to either the server’s local disk or to a network storage location. The function might also send email, write to a database, or store something on a FTP server. A single MediaScript function can be arbitrarily complex, and can therefore do almost anything that can be conceived of.

The following properties can be specified in a MediaRich request. Many of these are optional. Which of these are required and which are optional is a function of the particular client API being used.

Generator Name

Specifies the name or IP address of the Media Generator that will handle the request. If not supplied, the default is `localhost`.

Generator Port

Specifies the port number that the Media Generator is listening on. The default value is 9877, the standard Media Generator port number.

Script Name

Specifies the name of the script file that contains the function to be executed. Unless a file system is specified, the path will be interpreted relative to the *MediaRich Scripts* directory (see “File Systems” on page 39 for a description of how to specify file systems). In simple cases where a function execution is not required, a path to an image file can be supplied instead. In this case, the path is interpreted relative to the *MediaRich Media* directory. This property has no default value and so must be specified explicitly under all client APIs.

Function Name

The name of a function to call within the specified script file. If no function name is supplied, the Media Generator will default to calling a function named `main()`.

Function Arguments

A comma-separated list of arguments to the called function. Only numeric, boolean, and string literals are allowed in the argument list. Strings must be enclosed in quotation marks ("). If not supplied, the default value is an empty argument list.

Request Parameters

Request parameters are a list of named values that influence how the requested operation is performed. There are two types of request parameters: System Parameters and Script Parameters.

- System Parameters explicitly affect the behavior of the MediaRich platform. They are used for controlling the MediaRich cache and performing common image processing operations like scale and crop without requiring additional scripting. The names and formats of these parameters are detailed in “Post-Processing Parameters” on page 27 and “Cache Control Parameters” on page 34.

> ***Note:*** System Parameters names are reserved for explicit MediaRich operations and should not be used as the names of script parameters, irrespective of the API used to invoke the script. System parameters may be used to perform their defined operations from any of the APIs. Such operations are performed on the image saved by the script after the script has executed.

- Script Parameters are parameters that are ignored by MediaRich, but may be accessed by the script being executed. Functions access these parameters using the global request object’s `getParameter()` method (see “getParameter()” on page 58. Script Parameters serve a purpose similar to Function Arguments. A script may use either or both mechanisms.

HTTP API

The HTTP API is simply the API used by web browsers to talk to web servers. It involves sending a URL to the server and receiving back a standard HTTP response. The response will indicate if the request was performed successfully. If it was, the response can contain the results of the request, which might be an image (a JPEG or GIF file, for example), or a Web page or XML document.

MediaRich acts like a normal web server, and can therefore process HTTP requests. As with any HTTP request, the URL passed to MediaRich must be well-formed to describe the operation that the client would like MediaRich to perform. URLs that are properly formed to specify a MediaRich request are called MediaRich Resource Locators, or *MRLs*.

In HTML web pages, the `<img>` tag is used to insert a graphic or photographic image directly into the flow of text and other images. Traditionally, the `src` attribute of the `<img>` tag is a Uniform Resource Locator (URL) which points to a static image file. Using MediaRich, the URL is replaced with a MediaRich Resource Locator (MRL) that references a script function and contains parameters that are used to dynamically generate an image. MRLs can be static strings, or they can be generated automatically by client-side scripting languages such as VBScript or JavaScript or application server languages such as ASP or JSP.

Enabling the HTTP API

The HTTP API is only available if your MediaRich license includes the Web Server extension and MediaRich is installed behind a Web server. This is determined by the kind of MediaRich license that is installed. Refer to the *MediaRich CORE Installation and Administration Guide* for more information.

No \MediaRich\-specific modules or libraries are required on the client machine or in the client development environment. Any client that can make standard HTTP requests can talk to MediaRich using the HTTP API.

Note: The HTTP API is not available for the standard MediaRich for SharePoint product. Contact Equilibrium if you require the HTTP API with this product.

Working with MRLs

A MediaRich Locator, or *MRL*, is essentially a URL that points to a MediaScript and includes script parameters. When the request is made to a MediaRich server, the server executes the script with the specified parameters if that image is not already generated.

Note: Some caching mechanisms, such as Inktomi and AOL, will not cache a URL that contains a `?`. You can replace the `?` with a `/` in any MRL to avoid these problems.

MRL Format

A MRL is simply a URL of a specific format that MediaRich understands. Here is the general form of a MRL:

```
http://<server addr>/mgen/<script or image path> ?<param>=<value>&<param>=<value>...
```

A MRL string is formed by concatenating the following elements:

- The standard HTTP URL prefix `http://` - it can also use `HTTPS` if the MediaRich Web Server has been configured for SSL
- The name or IP address of the MediaRich server - this can include the server port number
- The value of the global `ImageServerRoot` property - the usual value for the image server root is `/mgen`
- A forward slash (`/`) separator
- The path to the MediaScript script file to execute or to a static image on the MediaRich server
- If additional request properties and/or parameters are included, they must be separated from the script path by a question mark (`?`) or forward slash (`/`). While the question mark is normally used, the forward slash may be required for some Web caches to cache MRLs.
- Zero or more properties and/or parameters of the form `name=value`, separated by the ampersand character (`&`).

As with any URL, all of the elements must be properly URL-encoded.

MRL Properties

MRLs can include properties (after the `?`) to define the name of the function to call and its arguments.

The function name is specified by:

```
f=<function name>
```

The function argument list is specified by:

```
args=<arg1>,<arg2>,...
```

Where the arguments are numeric, boolean, or string literals. String literals must be enclosed by quotation marks (").

MRL Parameters

Both System and Script parameters can also be included at the end of a MRL in the same way as the `f` and `args` properties. For more information about System parameters, see "Post-Processing Parameters" on page 27. Script parameters (those not understood by MediaRich) can be used by script developers for any purpose they define.

MRL Example

Suppose you have the following:

- A MediaRich server named www.mediarich.com
- The default image server root (`mgen`)
- A script in the MediaRich *Scripts* directory named *thumbnail.ms*
- A function in thumbnail.ms named `catalogThumb` that takes three arguments: the name of an image to thumbnail, the width of the thumbnail image, and the height of the thumbnail image

If you want to create a 100 pixel by 80 pixel thumbnail of an original named *bike.tif*, the MRL would be:

```
http://www.mediarich.com/mgen/thumbnail.ms?f=catalogThumb&args=%22bike.tif%22,100,80
```

Note: The quotation marks enclosing `bike.tif` are URL-encoded as `%22`.

The script function `catalogThumb` should have a definition like the following:

```
function catalogThumb(fileName, width, height)
{
// create a media object.
var m = new Media();
// load the file specified by the first argument.
m.load(name @ "myfiles:" + fileName);
// use the second and third arguments to scale the image.
m.scale(xs @ width, ys @ height);
// and return the result as a jpg image.
m.save(type @ "jpeg");
}
```

Working with SEOs

A Search Engine Optimized *MRL* or *SEO*, is an MRL format specifically created to be compatible with the common expectations of a referenced image resource for use on the web. An SEO is a reformatted MRL that allows for the much the same parameters and control over the resulting media object as an MRL, with only a change in the format of the resource call. For more on MRL's and how they function see Working with MRL's.

The SEO format is a more restrictive format than an MRL, and is intended as a replacement for media objects that you wish to have affected by Search Engine Optimization, and thus should be used where appropriate. Some MediaScripts may require modification to work within the restrictions of an SEO.

Some limitations of SEO’s are as follows:

- Parameters are positionally indexed - does not support argument name calls in the SEO
- SEO does not support post-process parameters
- SEO does not support ‘text strings’ and other such inputs as parameters

SEO Format

An *SEO* is simply a reformatted MRL, with a specific format and specific keys that MediaRich understands. The MediaScript resource is unchanged by the use of the SEO and thus depending on the script may be called by either MRL or SEO.

Here is the general form of an SEO:

```
http://<server addr>/mgen/ss/<script path>/<function name>/<param value>/<param
value>/.../nc(0/1)/<image path>/<image name>
```

An SEO, like an MRL string, is formed by concatenating the following elements:

- The standard HTTP URL prefix `http://` - it can also use `HTTPS` if the MediaRich Web Server has been configured for SSL
- The name or IP address of the MediaRich server - this can include the server port number
- The value of the global `ImageServerRoot` property - the usual value for the image server root is `/mgen`, followed by a forward-slash (`/`) separator
- The specific key '`ss`' to indicate this is an SEO formatted request, followed by a forward-slash (`/`) separator
- The path to the MediaScript script file to execute on the MediaRich server, followed by a forward-slash (`/`) separator
- The function name of the MediaScript function to be called, followed by a forward-slash (`/`) separator
- If additional request parameters are included, they must be separated by forward-slash (`/`) separator. These parameters are positionally indexed and must align with the order the MediaScript file expects the values

- The specific key '`nc(0/1)`' to indicate the use of MediaCache with the output file, followed by a forward-slash (`/`) separator. '`nc0`' indicates the use of the MediaCache features, '`nc1`' indicates MediaRich will not make use of the MediaCache and will regenerate the result file each time the SEO is called
- The path to the static image on the MediaRich server to be used as a source, followed by a forward-slash (`/`) separator
- The name of the resulting image generated by the MediaScript

As with any URL, all of the elements of the SEO must be properly URL-encoded.

SEO Example

Let's convert the existing MRL example to an SEO.

Suppose you have the following:

- A MediaRich server named www.mediarich.com
- The default image server root (`mgen`)
- A script in the MediaRich *Scripts* directory named *thumbnail.ms*
- A function in thumbnail.ms named `catalogThumb` that takes three arguments: the name of an image to thumbnail, the width of the thumbnail image, and the height of the thumbnail image

If you want to create a 100 pixel by 80 pixel thumbnail of an original named *bike.tif*, the SEO would be:

SEO:

```
http://www.mediarich.com/mgen/ss/thumbnail.ms/catalogThumb/100/80/nc1/bike.tif/bike_
thumb.jpg
```

MRL for reference:

```
http://www.mediarich.com/mgen/thumbnail.ms?f=catalogThumb&args=%22bike.tif%22,100,80
```

.NET API

The .NET API, along with the Java API, is the most powerful of the MediaRich APIs. It provides a set of class (object) definitions that allow complex requests to be sent to MediaRich. Such requests can included embedded media files (images, movies, etc.) as well as MediaScript code to be run on the server. Likewise, nearly anything may be sent back to the client by the server in the response and retrieved using this API, including any number of resulting media files.

Note: The .NET and Java APIs are very similar. They each provide the same set of classes and capabilities, and we endeavor to make them identical in syntax and convention whenever possible.

Using the .NET API

All of the materials necessary to develop against the MediaRich .NET API are installed on the server during the normal Windows MediaRich server installation process. In the default case, they are installed at the following location:

```
C:/Program Files/Equilibrium/MediaRich All Media Server/MediaRich APIs/.NET API
```

To use the .NET API, add a reference to the *Equilibrium.MediaRich.Util.dll* file to your C# or managed C++ project. Alternatively, you may install the *Equilibrium.MediaRich.Util.dll* file into the global cache by dropping it into the *assembly* folder in the system directory.

There is additional information about the .NET API in the file *MediaRich API.chm*. Refer to this resource for more details about using the API.

The *Sample* directory contains a sample client application that demonstrates the use of the API. Refer to the *ReadMe* file in that directory for information about building and running the sample.

.NET API Example

The following example illustrates the basic use of the .NET API in C#:

```
// Connect to the MediaRich Server
MGConnection connection = new MGConnection("mymrserver.somecompany.com");

// Create the request
MGRequest request = connection.CreateRequest();
request.SetScript("sample:/sampleScript.ms");
request.SetFunction("scale");
request.SetParam("amount", "0.5");

// Commit the request
MGResponse response = request.Commit();

// Retrieve the response content
if (response.HasContent)
{
int respLength = response.ContentLength;
byte[] respData = new byte[respLength];
response.ReadContent(respData);
```

Java API

The Java API, along with the .NET API, is the most powerful of the MediaRich APIs. It provides a set of class (object) definitions that allow complex requests to be sent to MediaRich. Such requests can included embedded media files (images, movies, etc.) as well as MediaScript code to be run on the server. Likewise, nearly anything may be sent back to the client by the server in the response and retrieved using this API, including any number of resulting media files.

Note: The Java and .NET APIs are very similar. They each provide the same set of classes and capabilities, and we endeavor to make them identical in syntax and convention whenever possible.

Using the Java API

All of the materials necessary to develop against the MediaRich Java API are installed on the server during the normal MediaRich server installation process. On a Macintosh server, they are installed at the following location:

```
/Applications/Equilibrium/MediaRich All Media Server/MediaRich APIs/Java API
```

On a Windows server, they are installed at the following location by default:

```
C:/Program Files/Equilibrium/MediaRich All Media Server/MediaRich APIs/Java API
```

On a Linux server, they are installed at the following location by default:

```
/usr/lib/Equilibrium/MediaRich_Media_Server/MediaRich APIs/Java API
```

To use the Java API, add the *MediaRichAPI.jar* file to your Java classpath during compilation of your client code. You must also make the contents of this file available at runtime by either incorporating the contents of the file into your own .jar file or by referencing it in your classpath.

The Java API documentation is supplied in standard javadoc form, in the *javadoc* directory.

The *Sample* directory contains a sample client application that demonstrates the use of the API. Refer to the *ReadMe* file in that directory for information on building and running the sample.

Java API Example

The following example illustrates the basic use of the Java API:

```
// Connect to the MediaRich Server
MGConnection connection = new MGConnection("mymrserver.somecompany.com");

// Create the request
MGRequest request = connection.createRequest();
request.setScript("sample:/sampleScript.ms");
request.setFunction("scale");
request.setParam("amount", "0.5");

// Commit the request
MGResponse response = request.commit();

// Retrieve the response content
if (response.hasContent())
```

```
{
int respLength = response.getContentLength();
byte[] respData = new byte[respLength];
response.readContent(respData);
}
```

Web Services API

The MediaRich server can be integrated into any stand-alone or web application through the MediaRich .Net Web Service. Web services use widely implemented standards such as XML and HTTP to support remote procedure calls across many languages and platforms.

Installing the MediaGenWebService

The MediaRich Web Service is implemented as an ASP.NET Web application. Before clients can use the Web Service, it must be installed in IIS using the MediaRich Installer. To install the MediaGenWebService, run the installer, select only the MediaGenWebService module, and complete the installation.

After the Web service is installed, you can view the available methods by browsing http://<host_name>/MediaGenWebService/MediaGenWebService.asmx.

Using MediaGenWebService

The MediaGenWebService client contains methods used to create and execute requests. This allows a request to be formed and forwarded to the MediaRich server via a Web method, providing the framework for building transaction-based services for various media types.

Using this service to create and execute requests requires the following:

- Create a web service
- Create a parameters list that is appropriate for the MediaScript that will be called
- Call one of the MediaGenWebService execute methods
- Handle the response

Creating the Web Service

The first step in using the MediaGenWebService is to add a web reference to it in your C# project. Create a Web service object similar to the following:

```
//create the web service
localhost.MediaGenWebService mgws = new
localhost.MediaGenWebService();
```

When you have created the web service object, you can apply other methods to it to perform MediaRich requests.

`MediaGenWebService()` is the default constructor.

MediaGenWebService ExecuteScriptCache Method

Use this method to call scripts that cache their responses. It returns a string containing a path to the image.

Syntax

```
string ExecuteScriptCache(string scriptName,
string functionName,
string[] parameters)
```

Parameters

`string scriptName` - indicates the script to execute.

`string functionName` - indicates the function in the MediaScript to execute.

`string[] parameters` - represent the array of strings that you would like to pass to your script. Each string must be of the form `name=value`. This is equivalent to everything that you would normally pass on a MRL after the function name.

MediaGenWebService ExecuteScriptStream Method

Use this method to call scripts that stream their responses. This method returns an array of bytes. The mime-type is an `out` parameter that you use to determine the type of image that the bytes constitute.

Syntax

```
byte[] ExecuteScriptStream(string scriptName,
string functionName,
string[] parameters,
out string mimeType)
```

Parameters

`string scriptName` - indicates the script to execute.

`string functionName` - indicates the function in the MediaScript to execute.

`string[] parameters` - represent the array of strings that you would like to pass to your script. Each string must be of the form `name=value`. This is equivalent to everything that you would normally pass on a MRL after the function name.

`out mimeType` - an output parameter that will be populated with the mime-type of the bytes returned.

MediaGenWebService ExecuteScript Method

This method may be used with scripts that cache or stream their responses. If the response is cached, the `out` parameter path will indicate where. If the response is streamed, the `out` parameter mime-type will indicate the mime-type and the `out` parameter buffer will contain the data.

Syntax

```
int ExecuteScript(string scriptName
string functionName,
string[] parameters,
out string path,
out string mimeType,
out byte[] buffer)
```

Parameters

`string scriptName` - indicates the script to execute.

`string functionName` - indicates the function in the MediaScript to execute.

`string[ ] parameters` - represent the array of strings that you would like to pass to your script. Each string must be of the form `name=value`. This is equivalent to everything that you would normally pass on a MRL after the function name.

`out string path` - returns a file path to the image.

`out string mimeType` - an output parameter that will be populated with the mime-type of the bytes returned.

`out byte[ ] buffer` - returns the image as an array of bytes.

Example

This example shows how a user would use the *MediaGenWebService* after adding a Web reference to it in a C# project.

```
<code>
try
{
//init
int i = 0;
string mimeType = "";

//create the web service
localhost.MediaGenWebService mgws = new localhost.MediaGenWebService();

//create parms string[]
parms = new String[3];
parms[0] = "args=200";
parms[1] = "nc=1";
parms[2] = "text=" + i.ToString();

//call the web service
byte[] buf = mgws.ExecuteScriptStream("eq/test4.ms", "foo", (string[])parms, out
mimeType);

//write the image to disk string
imgPath = "chewie" + i.ToString() + ".jpg";
FileStream fs = new FileStream(imgPath, FileMode.OpenOrCreate);

BinaryWriter w = new BinaryWriter(fs);
w.Write(buf); w.Close();
}
catch(Exception ex)
{
}
</code>
```

Note: There are several System parameters that can be used for special situations. For more information on System parameters see "Post-Processing Parameters" on page 27.

COM Client API

The MediaRich server can be integrated into any stand-alone or Web application through the MediaRich COM client. Examples of languages that support a COM binding include JavaScript, VB, VBScript, C, C#, and Perl.

Registering the MediaGenClient DLL

The MediaGenClient DLL (`MediaGenClient.dll`) is installed when you install MediaRich. It is located in the `\Equilibrium\MediaRich All Media Server\MediaRich APIs\COM API` directory.

To register the MediaGenClient DLL:

1. Copy the MediaGenClient.dll to the machine where you intend to run the COM API.
2. Open a command prompt.
3. Navigate to the directory where the copied MediaGenClient.dll is located.
4. Enter the following command:

   ```
   regsvr32 MediaGenClient.dll
   ```

This should register the MediaGenClient.

Important: Upgraded systems may need to repeat part of this process that was performed with the previous version of MediaRich.

Using the MGClient COM Interface

Using the MGClient COM interface to create and execute requests requires the following:

- Create an instance of the MGClient object
- Add the appropriate parameters for the MediaScript that will be called
- Call the `ExecuteScript()` method

If the MediaGenerator is streaming the data back to the client, write the return data to a location of your choice by calling `SaveBuffer()`.

If the MediaGenerator is returning cached data, examine the `Path` property for the file path to the returned data.

Creating the MGClient Object

Before issuing requests with the MGClient object, you must instantiate one. In order to do that you will have to specify the ObjectID, which has the form `servername.typename`. The ObjectID for the MGClient object is `MediaGenClient.MGClient`.

For example, if you were instantiating the MGClient object in JavaScript, you would invoke the ActiveXObject constructor and pass in the ObjectID:

```
var mgc = new ActiveXObject("MediaGenClient.MGClient");
```

When you have created a MGClient object, you can access its properties and call its methods to create and execute a request.

Creating and Executing a Request

When you have created a MGClient object, you formulate the request by setting properties of the client object and calling the object’s `setParameter()` method. The amount of information you must provide in the request will depend upon the function you are calling. Most properties have default values. Only the `ScriptName` parameter is required in all requests.

When the request has been properly formulated, a call to the `ExecuteScript()` method will dispatch the request to the MediaGenerator.

Request parameters

The parameters that are passed to the MediaScript function are managed with the methods provided by the MGClient:

- `GetParameter()`
- `SetParameter()`
- `RemoveParameter()`

There are several System parameters that can be used for special situations. For more information see “Post-Processing Parameters” on page 27.

MGClient Properties

The MGClient object supports the following properties:

Property	Type	Description
`HostName`	Read/write	Allows the user to set the hostname of the MediaRich server to be accessed. The default is `localhost`.
`Port`	Read/write	Allows the user to set the connection port on the MediaRich server. The default is `9877`.
`ImageServerRoot`	Read/write	Allows the user to set the imageServerRoot to correspond with the MediaRich server being accessed. The default is `/mgen`.
`ScriptName`	Read/write	Allows the user to set the name of the script to execute. This property is required — it has no default value.
`FunctionName`	Read/write	The function name to execute in the MediaScript file. The default is `main()`.
`MimeType`	Read-only	If this property is set after calling the `ExecuteScript()` method, it indicates that the server has sent back a binary response and this is its mime type.
`Buffer`	Read-only	Access this property to get the raw data returned by the MediaGenerator.
`BufferAsString`	Read-only	Access this property to get the data returned by the MediaGenerator as a string. If the MIME type of the data begins with something other than `text/` you will get `NULL/Undefined/Nothing`. The data returned must also be UTF-8 encoded (since ASCII is a subset of UTF-8, it is also OK).
`Path`	Read-only	If this property is set, the server caches the response and returns the path to the cached file.
`Timeout`	Read/write	Allows the user to set the timeout for socket send and receive operations. The default is `-1` (the same as the system's default time-out).

MGClient Methods

The MGClient object supports the following methods.

`void ExecuteScript()`

Sends a request to execute a script to the MediaGenerator. If there is an error an error will be thrown. If the request succeeds and a binary response has been streamed, the `MimeType` property will be set appropriately and the user may call `SaveBuffer()`. If the request succeeds and the response has been cached on the server the `Path` property will be set.

> ***Note:*** Each time this function is called, it resets the `MimeType`, `Buffer`, and `Path` to `NULL/Undefined/Nothing`.

`string GetParameter(string key)`

Gets the value of a parameter by key from the parameters collection.

`void SetParameter(string key, string value)`

Sets the value of a parameter by key from the parameters collection.

`string RemoveParameter(string key)`

Removes a parameter by key from the parameters collection. Returns the value of the removed key.

`void SaveBuffer(string path)`

Saves the response buffer that was streamed back from the server as the result of an `ExecuteScript()` call to disk. `Path` may be any valid win32 path.

The MediaGenerator Response

After the `ExecuteScript()` method is invoked the MediaGenerator will respond. Responses fall into two broad categories, error and success. Success responses may be further subdivided into cached and streamed responses.

If an error response is received, the MGClient object will produce an error most natural for the language which instantiated the object. In most languages that means the MGClient object will throw an error which may be handled with a try catch block. In languages that don't support try/catch semantics an error will be returned.

If a success response is received, the read-only `MimeType` and `Path` properties should be accessed to determine the success of the response. If the `MimeType` is populated then the client has received a binary stream, which can be written to the disk by calling the `SaveBuffer()` method. If the `Path` property is populated then the MediaGenerator has cached the response and the value of the `Path` property will tell you where.

In general the user-defined MediaScript function decides what the script will return and consequently what sort of response the MGClient object will receive. If the user-defined MediaScript function caches the response, you can override that decision when using the MGClient object by setting the System parameter `nc` to `1`. For more information about how to write and control what is returned from a MediaScript see “MediaScript Response Types” on page 61.

MGClient example

The following C# code demonstrates making a MediaRich request that returns an image.

```
using System;
using MEDIAGENCLIENTLib;

public class TestCom
{
public static void Main(string[] args)
{
try
{
// Create a MGClient object
MGClient mgc = new MGClient();

// Set the Media Generator to talk to
mgc.HostName = "Dev2";

// Set up the request (these are Script
// parameters)
mgc.ScriptName = "eq/demos/colorize/Color.ms";
mgc.SetParameter("img", "shoes.psd");
mgc.SetParameter("width", "400");
mgc.SetParameter("height", "100");
mgc.SetParameter("color", "0x00ff00");

// Cause the result to be spooled back to us
// instead of being put in the cache (this is
// a System parameter)
mgc.SetParameter("nc", "1");

// Execute the request
mgc.ExecuteScript();

// Save the result to disk
mgc.SaveBuffer("shoes.jpg");
Console.WriteLine("Got shoes.jpg");
}
catch (Exception e)
{
Console.WriteLine(e.ToString());
}
}
}
```

Using Request Parameters

By applying an operation at the request level, you can use a single MediaScript file multiple times, which reduces the number of files to manage and update. For example, using the `is` parameter in an MRL, you can use a single MediaScript file to create a thumbnail image, a full-sized image, and a cropped and enlarged version. More importantly, by updating that one MediaScript file, you automatically update all of the images that call it.

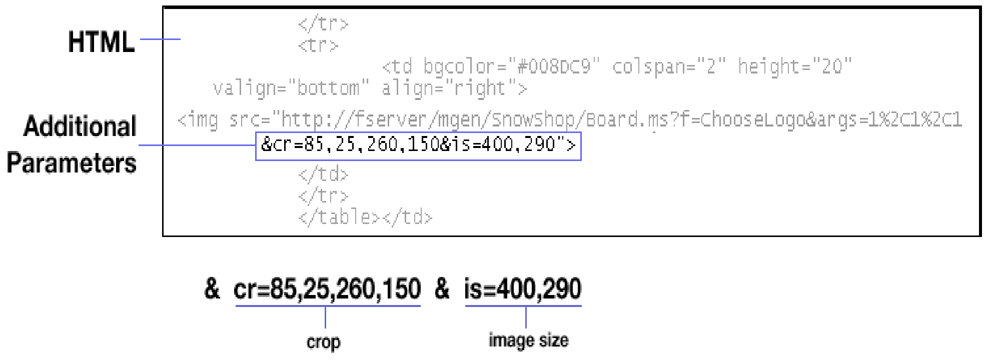
You can use post-processing arguments to perform simple operations on a source image without writing a MediaRich script. This is why the script path in a MediaRich request can be replaced with an image path. If the supplied path is an image path rather than a script path, then that image is simply returned to the caller. The post-processing parameters are applied to that image before it is returned.

Additionally, you can use pre-process parameters to specify the loading of the file and other parameters to retrieve file information and set caching.

Post-Processing Parameters

If a MediaRich request will result in an image (such as a JPEG or GIF), you can add parameters to the request to apply post-processing on that image before it is returned to the caller. Available post-processing operations include cropping, scaling, and image type conversion. For example, you can achieve identical scaling results using the `is` parameter in the MRL as you can using the `scale()` function in a script.

The following illustration demonstrates the use of these additional parameters in an MRL. They are appended to the main body of the MRL after the arguments and use the same ampersand (`&`) separators.



```
            </tr>
            <tr>
                    <td bgcolor="#008DC9" colspan="2" height="20"
  valign="bottom" align="right">
<img src="http://fserver/mgen/SnowShop/Board.ms?f=ChooseLogo&args=1%2C1%2C1
        &cr=85,25,260,150&is=400,290">
            </td>
            </tr>
            </table></td>
```

This section describes the following parameters:

- The Crop Parameter
- The Image Size Parameter

- The Slice Parameters
- The User Profile Information Parameter
- The Convert Parameter
- The Zoom Parameter
- The Grid-Zoom Parameter

For additional MediaRich parameters that can be specified in a request, see "MRL Parameters" on page 14.

The Crop Parameter

You can use the crop (`cr`) parameter to apply a crpo operation to the image (generated by the MediaScript file) by specifying *x,y* coordinates (in pixels) for the top-left and bottom-right corners of the desired, resulting image.

Syntax

```
cr=top,left, bottom,right
```

Example

```
http://www.eq.com/mgen/makeimage.ms?args=1&cr=85,25,260,150
```

This MRL takes the image generated by *makeimage.ms* and crops it. The cropped image uses the coordinates 85, 25 as the new top-left corner, and 260, 150 as the new bottom-right corner.

Note: All coordinates are based on the original top-left corner being 0,0.

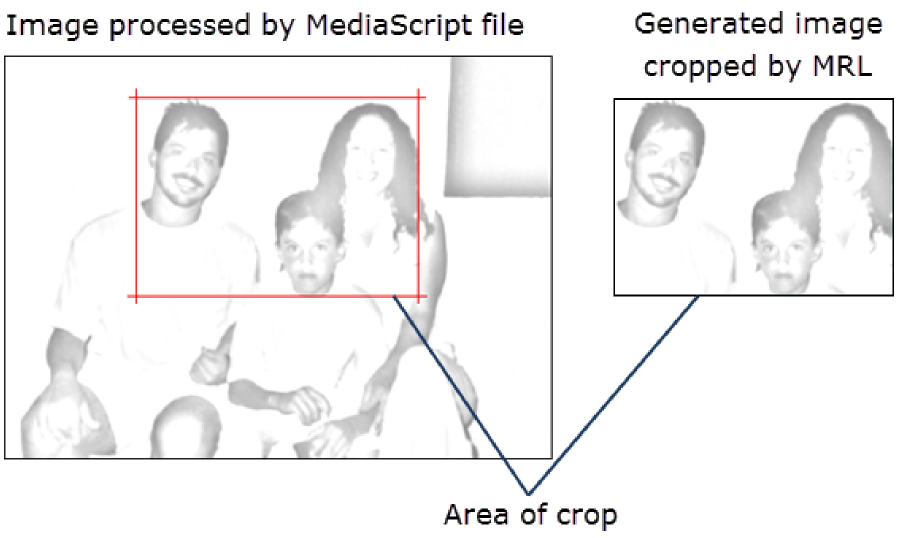


Image processed by MediaScript file
Generated image cropped by MRL
Area of crop

The Image Size Parameter

You can use the images size (`is`) parameter to specify the final image size and color (optional) of the response image in the MRL with the `is` parameter. You can also add a pad color to maintain the original aspect ratio; the pad color fills in the unused space.

For example, by specifying `is=300,200` MediaRich will scale the image to a width of 300 pixels and a height of 200 pixels.

Syntax

```
is=width,height,[hexadecimal value]
```

Example

```
http://www.mediarich.com/mgen/makeimage.ms?args=1&is=300,200
```

This MRL takes the image generated by *makeimage.ms* and resizes it from its original size to the size specified by the `is` parameter. In this case, the original image was 346 pixels by 255 pixels, and the resized image is 300 pixels by 200 pixels. Since 346 x 255 does not map exactly to 300 x 200, there is some distortion in the resulting image.

You can constrain the dimensions of the resized image to avoid distortion by specifying a hexadecimal RGB value for the pad color. As shown in the illustration below, the new image preserves the original aspect ratio, with the pad color filling in the areas that extend beyond the resized image.

346x225 image generated by MediaScript

300,200

Image resized to 300 x 200
with slight horizontal distortion

is=300,200,0x888888

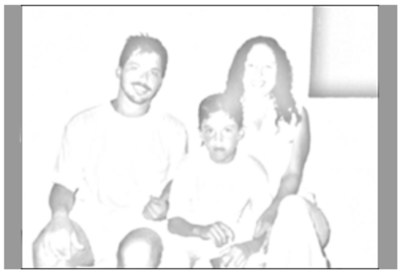


Image resized to 300 x 200
with pad to fill (no distortion)

The Slice Parameters

The MRL syntax for slice defines how an image is divided into equal pieces using columns and rows, and which piece of the image is returned. The slice table (`st`) specifies the number of rows and columns into which the image is divided. The slice position `sp` parameter specifies the piece of the sliced image to be returned to the browser (row, column).

Syntax

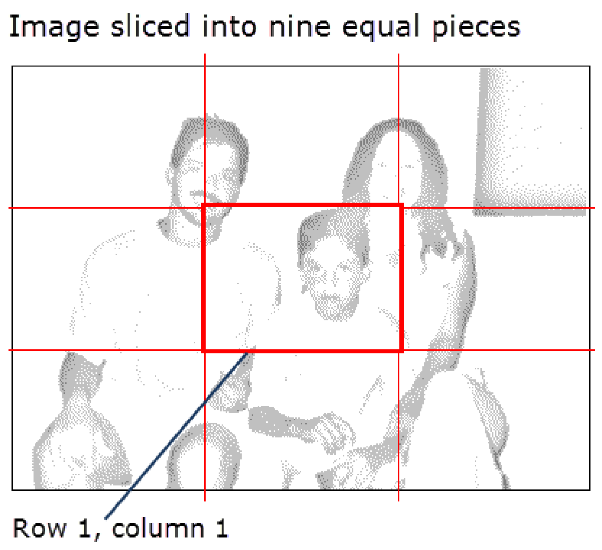
```
st=total_rows, total_columns
```

```
sp=specific_row, specific_column
```

Example

```
http://www.eq.com/mgen/makeimage.ms?args=1&st=3,3&sp= 1,1
```

This MRL takes the image generated by *makeimage.ms* and delivers a slice of that image based on information passed in the MRL. The `st` parameter slices the image into three rows and three columns. The `sp` parameter specifies that the portion of the image contained in row 1, column 1 be returned.



Note: The `sp` parameter is based on the upper left corner of the image being row 0, column 0.

The User Profile Information Parameter

Use the user profile `p` parameter to adjust image color and quality for a specific viewing device based on the quality and bit depth for that device.

Syntax

```
p=<device_name>
p=<quality:bit_depth>
```

Definitions for the profile parameters and the behavior of the p= operation are defined in the script ProfileScript.ms in the `MediaRichCore/Shared/Originals/Sys` directory. Definitions for additional profile types can be added to this script.

Note: The quality field applies only to JPEG images. If you specify a bit depth of 8 or less, the image is converted to GIF. JPEG supports only 24 bits per pixel or 8-bit grayscale, while GIF supports only 8-bit and smaller.

Example

```
http://www.mediarich.com/mgen/makeimage.ms?args=1&p=Palm
```

This MRL takes the image generated by the *makeimage.ms* and optimizes it for viewing on a Palm device.

The Convert Parameter

You can use the convert (`cvt`) parameter to specify the file type for the returned file.

Syntax

```
cvt=file type
```

The Zoom Parameter

Use the zoom `zm` parameter to zoom in or out and pan left, right, up, or down within an image.

Syntax

```
zm=width, height, zoomlevel, x-coordinate, y-coordinate
```

Example

The following MRL generates a scaled image of the top-left quadrant of the original image.

```
http://localhost/mgen/merchandizer/sample.jpg?zm=300,300,2,0,0
```

The Grid-Zoom Parameter

Use the `gz` parameter to apply a thumbnail grid as a navigational tool for displaying a zoomed portion of the image. The zoom level is one level higher than the displayed image.

Syntax

```
z=x-grid,y-grid, fraction, width, height, x, y
```

Example

The following MRL generates an unzoomed grid-zoom image with three grid spaces in each direction. The grid preview occupies 25% of the display area, which is 375 pixels wide and 450 pixels tall.

```
http://localhost/mgen/merchandizer/sample.jpg?gz=3,3,0.25,375,450,1,1
```

Pre-Process Parameters

The pre-process parameters are added to the load command, so if the particular format does not support them, they will just be ignored.

Since these parameters are used during load, all other post-processing parameters (`cr`, `is`, `cvt`, `st`, `sp`, `gz`, `zm`, `zmp`) can still be added to the MRL. In some cases (like PDF, that some browsers don't handle) it could be required to add "`&cvt=jpeg`" or “`cvt=png`”.

Page (pg)

Using this parameter, you can request a particular page from a multi-page file. It has an alias: fm (frame). It defaults to 1.

The following request will return page 5 of the file sample.pdf as a PNG image:

```
http://localhost/mgen/sample.pdf?pg=5&cvt=png
```

Resolution (res)

Use this parameter to request that when a particular file is rendered into a bitmap image, it will be rendered at particular resolution. It has an alias: dpi (dots per inch). It defaults to 300.

The following request will return page 5 of the file sample.pdf rendered at 200 DPI as JPEG image:

```
http://localhost/mgen/sample.pdf?pg=5&res=200&cvt=jpeg
```

.

Inquire and Load Parameters

Get Image Size (gis)

Use this to retrieve text with image size in the format: “width,height”. It takes a single parameter, which is the image path.

The following request:

```
http://localhost/mgen?gis=merchandizer:/sample_h.jpg
```

will return:

```
3517,2268
```

Merchandizer includes an example of its use (`MediaRich Merchandizer/Samples/Html/zoom_flash`), where a Flash Zoom object needs to know the size of the image in order to compute the area of the image to be requested from MediaRich at current zoom level and pan position.

Load Page with Resolution (lpr)

Given a multi-page document, it loads the specified page with specified resolution, saves it as a JPEG image in the cache, and returns a path to the cached image and total number of pages in the document. It takes three parameters (separated by commas):

- image path (required)
- page number (optional, defaults to 1)
- resolution in DPI (optional, defaults to 300)

The following request:

```
http://imac.home/mgen?lpr=merchandizer:/MRU_AdminGuide.pdf,1,300
```

will return:

```
OK:cache:/LoadPageRes/MRU_AdminGuide-page1-300dpi.jpg,56
```

At this point you should check that the first two characters are “OK” and if so, extract the path beginning at character 3, such as the following example:

```
if (httpRequest.responseText.substring(0,2) == "OK")
{
var fileInfo= httpRequest.responseText.substring(3).split(“,”);
var loadedPagePath = fileInfo[0];
var numPages = fileInfo[1];
}
```

Merchandizer includes an example of its use (`MediaRich Merchandizer/Samples/Html/zoom_flash/FlashZoomPageRes.html`), where JavaScript code loads requested pages from a mutli-page PDF file.

Cache Control Parameters

Parameters can be added to a MediaRich request that control how the results of the request will be cached, both within the MediaRich internal cache and in external Web caches.

This section describes the following parameters:

- The Time-To-Live Parameter
- The No Cache Parameter
- The Cache Control Parameter

For additional MediaRich parameters that can be specified in a request, see "MRL Parameters" on page 14.

The Time-To-Live Parameter

The Time-to-live (`tl`)parameter sets the lifetime for an image in the MediaRich cache. MediaRich keeps the generated image in the cache for the number of specified days, after which it is automatically cleared from the cache.

To prevent an image from being removed at the time specified in the global Time-to-live parameter, specify 0 days (`tl=0`) in the request.

Note: The Time-to-live parameter in a request overrides the global time-to-live setting in the MediaRich administration utility. For more information, refer to the *MediaRich CORE Installation and Administration Guide*.

Syntax

```
tl=days
```

Example

In the following example, the object will remain in the cache for seven days, and four hours:

```
http://www.medirich.com/mgen/makeimage.ms?args=1&tl=7%2E166
```

The No Cache Parameter

Specify the no cache (`nc`) parameter to 1 to disable caching, which causes any generated image to be streamed back to the caller instead of being placed in the cache. The `nc` MRL argument also accepts a comma-delimited list of parameters to ignore when computing the cache path. Changing the value of parameters included in this list has no effect on the cache path and does not result in generating a new image, but returns any cached image created using previous values of these parameters.

Syntax

```
nc=1
```

Example

If an image is generated with the following:

```
http://<server>/mgen/test.ms?text=%20test%20&date=%2004/01/2007%20&nc=date
```

The following subsequent request:

```
http://<server>/mgen/test.ms?text=%20test%20&date=%2004/02/2007%20&nc=date
```

returns the cached image for the previous request, generated using a date parameter of 04/01/2007.

This is intended to be used to pass administrative arguments on the MRL that are not directly involved in image generation, such as access tokens.

The Cache Control Parameter

The cache control (`c`)parameter allows standard cache control directives to be added to a MRL. These directives are used by external cache devices (for example, `c=no-cache`).

For more information, refer to the appropriate section of the WC3 specification for HTTP/1.1 (http://www.w3.org/Protocols/rfc2616/rfc2616- sec14.html#sec14.9).

Syntax

```
c=directive
```

Job Priority Parameter

Use the job priority (`jp`) to specify the prioritization of the job directly in the MRL. Possible values for this parameter are "low", "medium", and "high".

- low – a low priority job (currently equals a priority of 7)
- batch – a batch job (currently equals a priority of 5)
- normal – normal job (current equals a priority of 3)
- high – high priority job (currently equals a priority of 1)
- [0 thru 7] – sets the priority to that numeric value

If no `jp` value is specified, or if its value is not one of the valid values above, the job runs at “normal” priority.

CHAPTER 3

Using MediaScript

All MediaRich scripts 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 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 describes only those features unique to MediaScript; for a detailed description of the basic language, refer to the ECMAScript Language Specification (available at http://www.ecma-international.org/publications/standards/Ecma-262.htm) or one of the many available JavaScript references.

Chapter summary

The Media Object

The most important object in MediaScript is the Media object, which implements image processing features provided by MediaRich. A typical script creates one or more Media objects, then loads an image data from a file or using 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 “MediaScript Objects and Methods” on page 48.

Preprocessor Directives

Preprocessor directives are lines included in the code of programs that are not program statements but directives for the preprocessor. These lines are always preceded by a hash tag (#), are processed before the rest of the script, and can affect the way the script commands are interpreted.

The 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 by a backslash (\).

The following sections describe the preprocessor directives are used in MediaScript.

The MediaScript #include Directive

The `#include` directive allows other scripts to be included as if they were part of the original script. It takes a string (in quotes) representing the path of the script to include. The string must be enclosed in double quotes.

Syntax

```
#include "path to the included file"
```

Example

```
#include "scripts:mylibrary.ms"
```

To use the XmlDocument object, `#include` the *xml.ms*:

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

The MediaScript #link Directive

The `#link` directive allows to load precompiled objects, functions and data defined in a dynamic library (DLL). The directive takes a string (in brackets) representing the path of the file to load. When you use one of the libraries that ship with MediaRich (in the MediaRichCore/Bin/MediaEngine folder), you can omit the path (it defaults to “devices:”) and extension (it defaults to “.mdv”).

Syntax

```
#link <path to the library file>
```

Example

```
#link <EMailer>
```

Named Arguments

When reading the Media object reference in the next chapter, you will see that many of the object 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 command looks like the following:

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

Note: You cannot mix positional parameters and parameters passed with the @ argument in the same method call.

All 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 name is the argument name and the property 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 object. In this case, the MediaScript would be rewritten as:

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

File Systems

The MediaRich ECM 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 CORE 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 (Virtual File Systems)

MediaRich accesses files by defining a number of virtual file systems. A virtual file system (VFS) indicates the root of a file tree located somewhere on either the local file system or on the network. When referencing a file via a virtual file path, the virtual file system on which the file is located is specified by prepending the virtual file system name followed by a colon onto the path to that file, such as the following:

```
scripts:/path/script.ms
```

Virtual File System's name must start with an alphabetical character and contain only alphanumeric characters (punctuation marks are not allowed). Most file system specifiers are simply aliases for local or shared directories. The following built-in specifiers are located in the MediaRichCore/ directory:

File System Specifier	Default Location
`cache`	`Shared/Generated/MediaCache`
`dependencies`	`Shared/Generated/Dependencies`
`devices`	`Bin/MediaEngine`
`fonts`	`Shared/Originals/Fonts`
`glogs`	`Shared/Logs`

File System Specifier	Default Location
`logs`	`Shared/Logs/<hostname>`
`output`	`Shared/Generated`
`profiles`	`Shared/Originals/Profiles`
`read`	`(scriptPath);Shared/Originals/Media`
`results`	`Shared/Generated/MediaResults`
`scripterrors`	`Shared/Generated/ScriptErrors`
`scripts`	`(scriptPath);Shared/Originals/Scripts`
`sys`	`Shared/Originals/Sys`
`temp`	`Temp`
`write`	`(scriptPath);Shared/Originals/Media`

where `(scriptPath)` is the path to the currently executing script.

In case of multiple paths, MediaRich will first search all paths for an existing file and if it finds it, it will use it. If it does not find it, it will create it in the first path.

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 library	`devices`
Loading a color management profile	`profiles`
Saving a media file	`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.

Virtual File System

A Virtual File System (VFS) assigns a symbolic name to a directory on the MediaRich server. MediaRich accesses both local and remote storage solely through symbolic VFS names. Defining a VFS allows MediaRich to access a particular place on its local filesystem or on a network.

A VFS definition controls what MediaRich is allowed to do on that part of the filesystem. MediaRich is able to read from any defined VFS. A VFS definition can, however, either allow or deny write access to the portion of the filesystem it points to. Likewise, it can either allow or deny the execution of scripts from that portion of the filesystem.

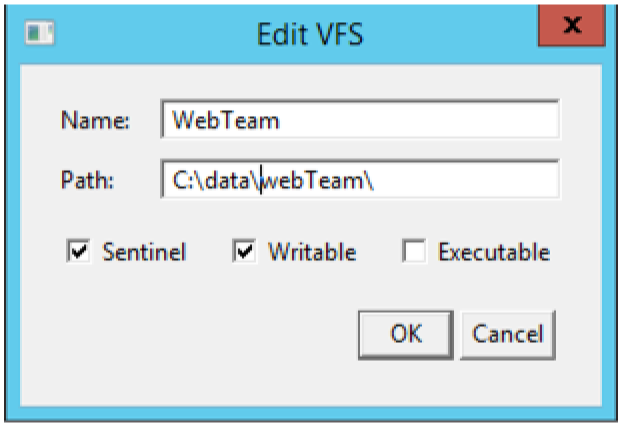
Using the MediaRich CORE Administration Utility, you can define any number of VFS directories for use in job submissions.

Note: If you expect to cause MediaRich to write files to a VFS directory, that directory must have its permissions set so that MediaRich has read and write permissions for all folders and sub-folders below the root of the VFS.

To define a Virtual File System:

1. In the MediaRich Server Options dialog, click the **Virtual Filesystems** tab.
2. Click the **New VFS** button.
3. In the Edit VFS dialog, enter a name for the VFS and its path.

   You can assign a name for the VFS according to your needs. You then specify the path of the directory you want to associate with this VFS. This path must be valid from the server's perspective, not from the perspective of how that same directory might be mounted on the network.



Name: WebTeam
Path: C:\data\webTeam\
☑ Sentinel ☑ Writable ☐ Executable

4. If you want to create a read-only directory that the MediaRich Server can use as a source directory but not as a save directory, clear the **Writable** checkbox.

   The **Writable** checkbox is selected by default, so if you want to create a standard read/write VFS directory leave this option selected.

   The **Executable** checkbox determines if scripts can be executed on this filesystem. This option is disabled by default. Enable this option if you want to store MediaScript script files on this filesystem that MediaRich can execute.

The **Sentinel** checkbox is required in order for the server to share files with clients. If you aren't sure if you need this option enabled, leave it enabled.

5. Click **OK** to set the new VFS definition.

To change or remove a VFS definition, select it in the MediaRich Server Options dialog and click **Edit** or **Remove**.

HTTP/FTP 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`. 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 45.

For `HTTPS`, simply specify port 443 in the path for your `Media.load()` op, like: `image.load (name @ "http://www.google.com:443/images/srpr/logo11w.png");`

Note: This only affects the Windows version because it uses the OS-provided HTTP services.

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 the *local.properties_user* 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_user* 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 `load()` and `save()` methods, that accept file paths as parameters. The paths passed as parameters to those methods 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/or 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://joe: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_user* 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=joe:tokyo
```

The specific properties that affect FSNet’s operation are described in the following subsections. As you read this information, remember that each of these properties can be specified such that they affect a single virtual filesystem, single protocol (`http` OR `ftp`), or all FSNet filesystems/both protocols (`http` AND `ftp`).

FSNet 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
```

FSNet 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
```

FSNet 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
```

Note: AWS S3 object storage and AZURE object datalake support is now available - See programmers guide for details on utilizing these new file access objects.

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 should use 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_user* 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
```

You can configure the following properties for FTP or HTTP:

Property	Usage
`Disabled`	Set to `1` to disable the file system, `0` to enable it
`Specifier`	This is the path specifier that denotes this file system.
`UserName`	Sets the default user name.

Property	Usage
`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` to break a lock on a cache file, if it fails to become available.
`RefreshInterval`	Specifies how often (in seconds) to check the Web for a newer file.

MediaScript 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 similar to 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 HTML errors are disabled, MediaRich will respond with a 500 Internal Server Error status code.

CHAPTER 4

MediaScript Objects and Methods

MediaScript includes a number of built-in objects that you can use when writing scripts.

Chapter summary

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.

Note: MediaRich reads up to 16-bit per channel, and automatically converts 16bit per channel down to 8-bit per channel before operations can be handled. The Photoshop reader also converts 24-bit per channel High Dynamic Range images to 8 bits internally. Additionally, 32 bits per channel, LAB and Pantone color space are currently not supported in 6.X MediaRich Server.

Non-Executing Media Process Parameters

All media transform commands can use the following 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 image (frame). MediaScript supports a “frames” parameter that can be included with all transform commands for processing specific frames within a multi-frame file.

`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

This script flips the 5th frame:

```
image.load(name @ "clock.gif");
image.flip(axis @ "horizontal", frames @ "5");
image.save(name @ "flip5.gif");
```

This script flips the 1st, 5th, 7th, 8th, 9th, and 10th frames:

```
image.load(name @ "clock.gif");
image.flip(axis @ "horizontal", frames @ "1,5,7-10");
image.save(name @ "multiflip.gif");
```

This script flips the 4th, 6th, 8th, and 10th frames (every 2nd frame):

```
image.load(name @ "clock.gif");
image.flip(axis @ "horizontal", frames @ "4-10-2");
image.save(name @ "flipskip.gif");
```

MediaScript Global Functions

MediaRich and the MediaScript language supports all the basic ECMAScript capabilities, including conditionals, variables, functions, and exception handling, as well as the proprietary image processing functions. There are a number of commands and format options specifically related to the 64-bit file size capabilities supported in MediaRich CORE 4.0.

For information on functions and objects not described in this guide, refer to the ECMAScript specification at http://www.ecma-international.org/publications/standards/Ecma-262.htm or to a JavaScript reference guide.

Note: The `error()` function is now deprecated. Use the standard JavaScript try..catch..finally and throw syntax instead.

Global Methods

- batch.setJobStatus()
- clearCached(mrl)
- COMCreateObject()
- getPropertyValue()
- getScriptFileName()
- print()
- rgb()
- stringTrimBoth()
- stringTrimEnd()
- stringTrimStart()
- version()

batch.setJobStatus()

Sets the status message for the currently executing script. Currently, this message is displayed in the Server Administration Server Status window, in the “Status” column, on the line associated with the worker process that is processing the job. Refer to the *MediaRich CORE Installation and Administration Guide* for information about the Server Status window.

Syntax

```
batch.setJobStatus(<string>);
```

Example

This function, by default, displays a job status message that counts from 1 to 10, sleeping for one second between updates to the message. The number of repetitions and the sleep time can be modified via MRL parameters.

```
function status()
{
// Determine the sleep time (in seconds)
var sleep = req.getParameter("sleep");
if (!sleep)
sleep = 1;
// Determine the iteration count
var count = req.getParameter("count");
if (!count)
count = 10;
// Do it!
for (var i = 1 ; i <= count ; i++)
{
batch.setJobStatus("Status: " + i);
System.threadSleep(sleep);
}
}
```

clearCached(mrl)

The `clearCached(mrl)` removes media specified by mrl from MediaRich cache.

Note: This method should not be confused with the clearCached method that is a member of the File Object e.g. `File.clearCached()` that method helps you manage the FSNet cache.

Syntax

```
clearCached(<mrl>)
```

Parameters

`mrl` - the full MRL that originated the request for the media object.

Example

```
clearCachedTest.ms
```

A new script to demonstrate and test the removal of a given item from the MediaScript cache.

Example of use:

```
http://localhost/mgen/clearCachedTest.ms?f=ClearCached&mrl=http://localhost/mgen/test
.ms
```

COMCreateObject()

Creates a COM object (Windows only).

Parameters

`progId` - specifies a string containing the friendly progID of the COM object.

getPropertyValue()

The `getPropertyValue()` function 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 this function, you can access properties in these files from within MediaScript.

Note: Properties are controlled by the MediaRich system administrator using the MediaRich Administration Utility. Refer to the *MediaRich CORE Installation and Administration Guide* for more information.

Syntax

```
getPropertyValue(<propertyName>);
```

Parameters

`propertyName – string –` property name in quotes. Property names consist of the filename in which the property exists (excluding the extension), a period (`.`), and the actual property name.

Note: Property name information is case-sensitive.

Example

To access the `LogLevel` property in the *local.properties* files:

```
var logLevel = getPropertyValue("local.LogLevel");
```

If *local.properties* (or *local.properties_user*) file includes the following line:

```
LogLevel=severe
```

variable logLevel will contain the string “severe”.

getScriptFileName()

The `getScriptFileName()` function returns the filename of the currently running script.

Syntax

```
getScriptFileName();
```

print()

The `print()` function prints the specified string to the MediaGenerator.log file with a severity level set by the property `local.PrintSeverity`.

Syntax

```
print(<string>);
```

rgb()

The `rgb()` function 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>);
```

stringTrimStart()

The `stringTrimStart()` function returns a string with all whitespace characters removed from the start of the string.

Syntax

```
stringTrimStart( <string> );
```

Parameters:

`string` - string from which whitespace is to be trimmed.

stringTrimEnd()

The `stringTrimEnd()` function returns a string with all whitespace characters removed from the end of the string.

Syntax

```
stringTrimEnd( <string> );
```

Parameters

`string` - string from which whitespace is to be trimmed.

stringTrimBoth()

The `stringTrimBoth()` function returns a string with all whitespace characters removed from both the end and start of the string.

Syntax

```
stringTrimBoth( <string> );
```

Parameters

`string` - string from which whitespace is to be trimmed.

version()

The `version()` function returns a string that is the current version of MediaScript.

Syntax

```
version();
```

Request and Response Global Objects

For every request, MediaScript creates a global HTTP Request object named *req* and a global HTTP Response object named *resp*.

The Request and Response objects do not need to be constructed. A static global instance of each is created for each MediaScript execution context.

Request Objects

You can use the request object to get request parameters, HTTP headers, and information about the MRL.

The `req` object includes the following methods:

- getBatchId()
- getBatchTempDir()
- getFileParamNames()
- getFileParamPath()
- getHeader()
- getHeaderNames()
- getJobId()
- getJobTempDir()
- getNotCached()
- getParameter()
- getParameterNames()
- getPath()
- getQueryString()
- getRequestURL()
- getScriptPath()

getBatchId()

The `getBatchId()` method returns an integer identifying the batch. A batch is a collection of jobs sent to the *MediaGenerator* through the .NET of Java APIs.

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. For a description of the MediaRich batch interface, see the .NET or Java API online documentation available in the *SDK* folder. For non-batch requests, this identifier is always `0`.

Syntax

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

Parameters

This function has no parameters.

getBatchTempDir()

The `getBatchTempDir()` method 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.

getFileParamNames()

The `getFileParamNames()` method returns an array containing the names of all file parameters. File parameters describe files sent to the MediaGenerator through the .NET or Java APIs for processing.

The names returned by this method can 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 or Java APIs. For more information, see the .NET or Java API online documentation available in the *SDK* folder.

Syntax

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

Parameters

This function takes no parameters.

getFileParamPath()

The `getFileParamPath()` method 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

Given that a file parameter is passed to a script with a parameter name of `testFile`, this file can be accessed within the following script:

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

getHeader()

The `getHeader()` method 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)
```

getHeaderNames()

The `getHeaderNames()` method 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)
```

getJobId()

The `getJobId()` method returns an integer identifying the current job.

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. For a description of the MediaRich batch interface, see ".NET API" on page 16 and "Java API" on page 17.

Syntax

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

Parameters

This function has no parameters.

getJobTempDir()

The `getJobTempDir()` method 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.

getNotCached()

The `getNotCached()` method returns `true` if this is a no-cache request.

Syntax

```
var notCached = req.getNotCached();
```

Parameters

This function takes no parameters.

getParameter()

The `getParameter()` method 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()

The `getParameterNames()` method returns an array of the names of all parameters specified on the MRL.

Syntax

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

Parameters

This function takes no parameters.

getPath()

The `getPath()` method returns the path previously set using `setPath()`.

Syntax

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

Parameters

This function takes no parameters.

getQueryString()

The `getQueryString()` method 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.

getRequestURL()

The `getRequestURL()` method returns the full MRL that originated the request.

Syntax

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

Parameters

This function has no parameters.

getScriptPath()

The `getScriptPath()` method 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"
```

Response Objects

You can use the response object to set the response contents, HTTP response headers, and status code.

Setting the Response Contents

There are several ways to set the response contents. The simplest is to use the Media object `save()` method without a name parameter, such as the following:

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

When the script terminates, the contents of the `img` object are sent back as a response. They are also 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 `setObject()` method. For example, the previous example could be rewritten as the following:

```
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 61.

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

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

By default the file is copied to the cache. You can also stream the contents back to the client or return only the file path by setting the response type to `RespType.Path`.

MediaScript 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.

Methods

The `resp` object includes the following methods:

- getHeader()
- getMedia()
- getMimeType()
- getObject()d
- getPath()
- getResponseType()
- getSaveParameters()
- getStatusCode()
- setHeader()
- setMedia()
- setMimeType()
- setObject()
- setPath()
- setResponseType()
- setSaveParameters()
- setStatusCode()
- write()
- writeLine()

setObject()

The `setObject()` method sets the response object, which must be a Media, TextResponse, or XmlDocument 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. Must be one of Media, TextResponse, or XmlDocument.

`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. This is the default if the nc=1 parameter is not specified.
- `RespType.Streamed` bypasses the cache and returns the response data directly to the filter. This is the default if nc=1 is specified.
- `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 64.

`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()d

The `getObject()` method 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()

The `setPath()` method 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 39 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 64.

Example

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

getPath()

The `getPath()` method returns the path of the current response Media object. The method takes no arguments.

Parameters

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

Syntax

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

setMedia()

The `setMedia()` method 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()

The `getMedia()` method returns the current response Media object.

Syntax

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

Parameters

This function has no parameters.

setResponseType()

The `setResponseType()` method 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()

The `getResponseType()` method returns the response type previously set by setPath(), setObject (), or setResponseType().

Note: If no response type is set, the method returns null.

Syntax

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

Parameters

This function has no parameters.

setSaveParameters()

The `setSaveParameters()` method 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 177.

getSaveParameters()

The `getSaveParameters()` method returns the current save parameters for the response Media object.

Syntax

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

Parameters

This function has no parameters.

setMimeType()

The `setMimeType()` method 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()

The `getMimeType()` method returns the response MIME type or, if not set, undefined.

Syntax

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

Parameters

This function has no parameters.

setHeader()

The `setHeader()` method 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()

The `setStatusCode()` method 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()

The `getStatusCode()` method returns the response status code.

Syntax

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

Parameters

This function has no parameters.

write()

The `write()` method 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()

The `writeLine()` method 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.

Important: Loading, modifying, and saving very large image files can result in errors or crashes when the system cannot accommodate these files. If this occurs, you should first try adding more memory to your server. For more information, see "Memory Issues with Very Large Image Files" on page 362.

Also, when working with large images, please ensure your page file size is set to "System Managed Size" on the enabled drive and make sure the drive has enough space to contain it.

Media Object Class Methods

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

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

getFileInfo()

This `getFileInfo()` method returns information about the image or images contained in the specified file. It returns as much information as can be retrieved 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 varies from one file type to another. The only value that is guaranteed to be returned is the `Type` value, which is the name of the file type (such as TIFF or JPEG).

The following values are returned by most file types:

Value	Description
`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

The following values are typically returned by some file types:

Value	Description	Known file formats
`XDpi`	The horizontal resolution of the image(s)	
`YDpi`	The vertical resolution of the image(s)	
`Dpi`	The DPI information of the image(s)	BMP
`Frames`	The number of frames in the file	SWF and animations
`FrameRate`	The number of per second to be displayed	SWF and animations
`ResolutionUnits`	The number of per second to be displayed	SWF and animations

- `XDpi` - the horizontal resolution of the image(s)
- `YDpi` - the vertical resolution of the image(s)
- `Dpi` - the DPI information of the image(s) (for BMP files)
- `Frames` - the number of frames in the file
- `FrameRate` - the number of per second to be displayed (for SWF and animations)
- `ResolutionUnits` - (for SWF and animations)
- `XResolution` - (for SWF and animations)
- `YResolution` - (for SWF and animations)
- `Layers` - the number of layers in the file (for PSD and other files with layers)
- `Version and Type` - (for SWF and animations)

`Media.getFileInfo()` returns the following fields (all are text strings) in the info object for most LibreOffice-handled documents:

- `Title` - Title of the document
- `Author` - Original author
- `LastEditedBy` - Name of the person who last edited the document
- `Comment` - The comment area in the document properties
- `Generator` - Name of the application that created the document
- `CreationTime` - Time the document was originally created. In yyyy/mm/dd hh:mm:ss format
- `LastEditTime` - Time the document was last edited
- `LastPrintTime` - The the document was last printed
- `Keywords` - A list of keywords on a single line from the document properties

- `Statistics` - Miscellaneous statistics about the doc, like # of pages, # of cells, etc.
- `TemplateName` - The document template name

> ***Note:*** Not all documents have all of these fields, and in the case of the Statistics field, not all documents will return the number of pages or other information.

Many of the plain image formats read via LibreOffice (such as PhotoCD/PCD) do not return any useful information at all. Metadata is limited to text-based documents and Power Point presentations.

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");
}
```

Access individual properties using something similar to:

```
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.
  - `Media.FormatExifLoad` -> Use if the format supports loading Exif metadata.
  - `Media.FormatExifSave` -> Use if the format supports saving Exif metadata.
  - `Media.FormatIPTCLoad` -> Use if the format supports loading IPTC metadata.
  - `Media.FormatIPTCSave` -> Use if the format supports saving IPTC metadata.
  - `Media.FormatXMPLoad` -> Use if the format supports loading XMP metadata.
  - `Media.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 describing all 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()

The `getExtensionFromType()` method 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` or `tiff`).

getTypeFromExtension()

The `getTypeFromExtension()` method returns the Media type for the given extension.

Syntax

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

Parameters

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

Media Object Methods

The Media object is constructed using the `Media()` constructor.

Syntax

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

Methods

- addArgument()
- adjustHsb()
- adjustRgb()
- arc()
- blur()
- blurBlur()
- blurGaussianBlur()
- blurMoreBlur()
- blurMotionBlur()
- clone()
- collapse()
- colorCorrect()
- colorFromImage()
- colorize()
- colorToImage()
- composite()
- convert()
- convolve()
- crop()
- digimarcDetect()
- digimarcEmbed()
- digimarcRead()
- discard()

- drawText()
- dropShadow()
- ellipse()
- embeddedProfile()
- equalize()
- exportChannel()
- fixAlpha()
- flip()
- frameAdd()
- getAverageColor()
- 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()
- getResHorizontal()
- getResVertical()
- getSamplesPerPixel()

- getWidth()
- getXmlInfo()
- glow()
- gradient()
- importChannel()
- infoText()
- line()
- load()
- loadAsRgb()
- makeCanvas()
- measureText()
- noiseAddNoise()
- otherMaximum()
- otherMinimum()
- pixellateFragment()
- pixellateMosaic()
- polygon()
- quadWarp()
- rectangle()
- reduce()
- 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()

adjustHsb()

The adjustHsb() method alters the HSB levels of an image. It can be applied to images of all supported bit-depths.

Note: This function is "selection aware"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

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.

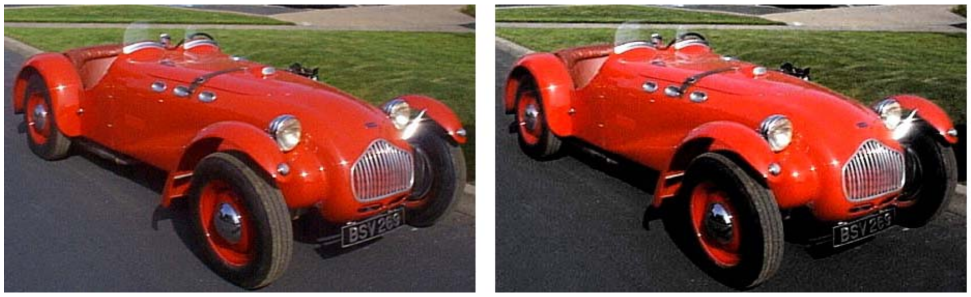
`Saturation` and `Brightness` - linear color values that set the base level for the saturation and brightness of the image.

`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()

The `adjustRgb()` method alters the contrast, brightness, and color balance of an image.

> ***Note:*** This function is "selection aware"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

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.

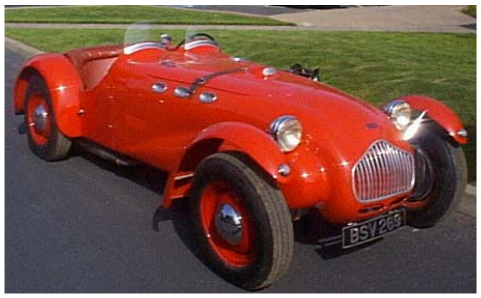
`Red`, `Green`, and `Blue` - adjust the brightness of each of the three color channels individually.

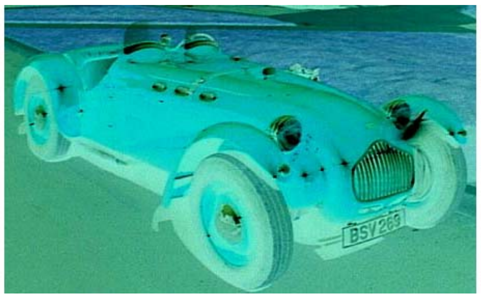
`Noclip` - when specified, brightness adjustments will avoid maxing-out (either high or low) the image by reducing the contrast accordingly. A contrast offset can be used to override this process.

`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()

The `arc()` method 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 @ <true, false>]
[Color @ <color in hexadecimal, rgb, or cymk>]
[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. This parameter supports a hexidecimal, RGB, or CMYK color specification:

- **hexidecimal** - color value expressed as a value from 0x000000 to 0xFFFFFF (RGB colorspace) or from 0x00000000 to 0xFFFFFFFF (CMYK colorspace)
- **RGB** - color value expressed as a value from 0 to 16,777,215
- **CMYK** - color value expressed as a value from 0 to 4,294,967,295

Colorspace

Always pass a color value appropriate to the colorspace. You can ensure this using the getPixelFormat() function in your script and then using different hexadecimal values for the RGB and the CMYK colorspaces in an IF/THEN construction. If getPixelFormat() returns "CMYK," use the CMYK value (0x plus eight more digits), and otherwise use the RGB value (0x plus six more digits).

`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 specify 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: The `Burn` option is 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`.

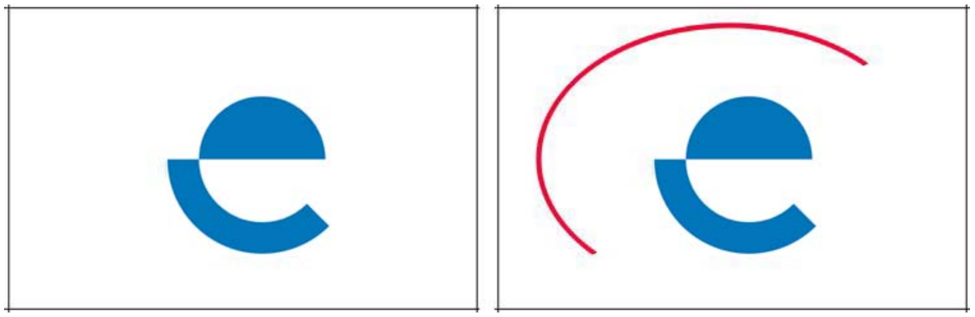
Important: If you are using smoothing for media that contains an alpha channel and you plan to save it to a format that does not support alpha channels, it is necessary to use convert() to remove the alpha channel before using this operation. Or, as an alternative, you can composite the modified image onto an opaque background before saving the image. Without this additional handling, the media will not look correct in a non-alpha file format.

`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()

The `blur()` method 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"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

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”—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see “selection()” on page 186.

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()

The `blurGaussianBlur()` method applies a Gaussian blur effect to the image.

Note: This function is “selection aware”—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

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()

The `blurMoreBlur()` method applies a similar but stronger blur effect as the `blurBlur()` function.

Note: This function is "selection aware"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

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()

The `blurMotionBlur()` method 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"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

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()

The `clone()` method 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()

The `collapse()` method 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 colorspace 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

The `collapse()` method supports the following parameters:

Parameter	Usage
`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.
`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`, ignores the alpha channel information when collapsing the image layers The default is `false`.
`VisibleOnly`	When set to `true`, includes only the layers designated as visible in the collapsed image The default is `false`.
`likePS`	When set to `true`, iperforms the collapse like Photoshop where he default background color is white, the alpha channel is removed, and the collapsed image is made opaque The default is `false`. **Note:** Because it uses a white background color, this parameter should be disabled for images that will later be used as a brush.

Loading with PreviewAlpha

The `previewAlpha` parameter is set to `true` by default for the `load()` method (unless `Layer` or `Track` are specified), which loads the Photoshop 2.5-era alpha channels. This setting promotes that background layer that has no true alpha to a foreground layer and uses this channel as the alpha. Upon collapse, this takes the alpha from layer 0 and layer 1 and mixes them to produce transparency all the way through the bottom of the image.

Fore more information about using the `previewAlpha` parameter, see “Photoshop-specific parameters” on page 145.

Collapsing named layers

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.”

Example

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

colorCorrect()

The `colorCorrect()` method transforms an image from a source colorspace to a destination colorspace. 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 *MediaRichCore/Shared/Originals/Profiles* directory and the system color profile directory if there is one. The *MediaRichCore/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 *local.properties* file can be modified to change the default profile directory. Refer to the *MediaRich Installation and Administration Guide* for more information.

Syntax

```
colorCorrect(
destProfile @ <"filename.icc">
[sourceProfile @ <"filename.icc">]
[intent @ <"rendering intent">]
[overrideEmbedded @ <true, false>]
[BlackPointCorrection @ <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: *MediaRichCore/Shared/Originals/Profiles*.

`BlackPointCorrection` - This parameter defaults to false (disabled). The black point correction will also not occur if the "intent" param is set to "Perceptual". It must be set to one of the other modes in order to see any difference when converting.

Note: You can modify the MediaRich server's *local.properties* file to change the default */Profiles* directory. Refer to the *MediaRich Installation and Administration 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 colorspace is affected. 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, see "MediaRich Color Management" on page 318.

Example

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

colorize()

The `colorize()` method changes the hue of the pixels in the image to the specified color.

Note: This function is "selection aware"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

Syntax

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

Parameters

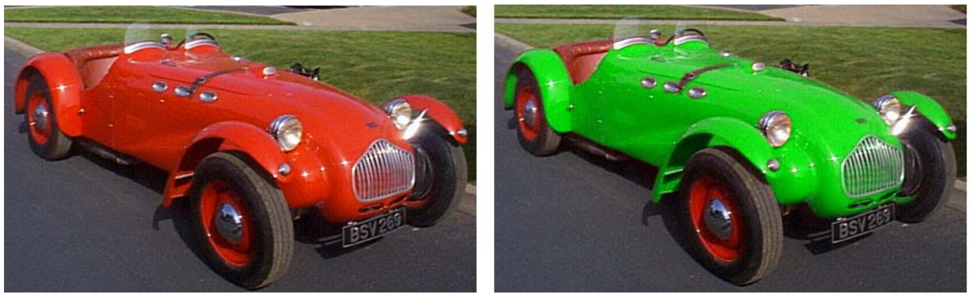
`Color` - specifies the color using its hexadecimal or rgb value. The best results will appear by creating or loading a selection first.

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 137.

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 `colorFromImage()` method converts the specified color 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, 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 86.

`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, see CHAPTER 9 , “MediaRich Color Management” on page 318.

colorToImage()

The `colorToImage()` Method converts the specified color 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 86.

`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, see CHAPTER 9 , “MediaRich Color Management” on page 318.

composite()

The `composite()` method 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”—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see “selection()” on page 186.

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>]
[PreserveAlpha @ <true, false>]
[IgnoreAlpha @ <true, false>]
[X @ <pixel>]
[Y @ <pixel>]
[HandleX @ <"left", "center", "right">]
[HandleY @ <"top", "middle", "bottom">]
[Tile @ <true, false>]
[Blend @ <"blend-type">]
);
```

Important: 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()" on page 88.

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`.

`IgnoreTransparency` - when set to true, the source is composited onto the target and all transparency information, via alpha channel or other forms of transparency, is ignored. The default is `false`. (This option was formerly known as `IgnoreAlpha`, and that name can still be used, but the behavior is as described above, with both alpha channels and all other forms of image transparency ignored.)

`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()

The `convert()` method 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 colorspace to RGB colorspace 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-24`, `RGBA-32`, `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`
- `CMYK -> CMYK-32`
- `CMYKA -> CMYKA-40`

> ***Note:*** Deprecated parameters include: `Grayscale`, `pal-8`, `15-bit`, `16-bit`, `16a-bit`, `24-bit`, `32-bit`.

MediaRich reads up to 16-bit per channel, and automatically converts 16-bit per channel down to 8-bit per channel before operations can be handled. The Photoshop reader also converts 24-bit per channel High Dynamic Range images to 8-bits internally. Additionally, 32 bits per channel, LAB and Pantone color space are currently not supported as of the writing of this documentation.

`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 137.

Example

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

convolve()

The `convolve()` method 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()

The `crop()` method 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 (see page 188), 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 colorspace), 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">]
[Relative @ <true, false>]
);
```

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. The `Relative` parameter effects how these values are interpreted.

`Relative` - Defaults to `true`, which preserves the standard behavior of cropping using coordinates passed (`Xo` and `Yo`)that are relative to the origin. Specify `false` to crop using absolute coordinates, which is particularly useful for tiled images.

`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 137.

`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 188.

`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()

The `digimarcDetect()` method detects a Digimarc watermark in the file.

Digimarc 4.0 is supported in MediaRich CORE 6.2.

Important: Contact a sales representative at Equilibrium if you do not have Digimarc Server Licenses and the necessary ID and PIN number to achieve this process. Both the Digimarc iKit from Equilibrium, as well as, Digimarc Server + Maintenance licenses are required for the Digimarc Imaging Server to operate using MediaRich Server. Equilibrium is an authorized Digimarc reseller and can fulfill all Digimarc Server needs. If you already have your Digimarc License directly with Digimarc, please call Equilibrium to purchase the Digimarc iKit that enables the automatic Digimarc processing to occur within a MediaRich MediaScript environment.

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 detected
Image.drawText(font @ "Arial", style @ "Bold", text @ "A DigiMarc watermark has been
detected!", size @ 20);
Image.save(type @ "jpeg");
}
```

digimarcEmbed()

The `digimarcEmbed()` method embeds Digimarc information in the specified image.

Digimarc 4.0 is supported in MediaRich CORE 6.2.

Important: Contact a sales representative at Equilibrium if you do not have Digimarc Server Licenses and the necessary ID and PIN number to achieve this process. Both the Digimarc iKit from Equilibrium, as well as, Digimarc Server + Maintenance licenses are required for the Digimarc Imaging Server to operate using MediaRich Server. Equilibrium is an authorized Digimarc reseller and can fulfill all Digimarc Server needs. If you already have your Digimarc License directly with Digimarc, please call Equilibrium to purchase the Digimarc iKit that enables the automatic Digimarc processing to occur within a MediaRich MediaScript environment.

Syntax

```
digimarcEmbed(
[Type @ <"type">]
[CreatorID @ <id 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>]
[TargetResolution @ <DPI amount>]
);
```

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 that uniquely identifies the creator of the image. The creator ID maps to a profile of the creator, at the Digimarc MarcCentre Web site. Valid for the following types: `basic`, `image`, `transaction`, and `copyright`.
`DistributorID`	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 the image has restricted use when set to `true`. The default value is `false` (empty). Valid for all types.

Parameter	Description
`CopyProtected`	Indicates that the 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.
`TargetResolution`	TargetResolution takes a value in DPI units to affect the size of the pixels Digimarc uses to embed the watermark. If this parameter is left off, it defaults to 100 DPI. The user can specify higher resolutions if they are watermarking print media, where having larger watermark pixels might be useful if the media gets rescanned.

Example

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

digimarcRead()

The `digimarcRead()` method returns Digimarc information embedded in the specified image.

Digimarc 4.0 is supported in MediaRich CORE 6.2.

Important: Contact a sales representative at Equilibrium if you do not have Digimarc Server Licenses and the necessary ID and PIN number to achieve this process. Both the Digimarc iKit from Equilibrium, as well as, Digimarc Server + Maintenance licenses are required for the Digimarc Imaging Server to operate using MediaRich Server. Equilibrium is an authorized Digimarc reseller and can fulfill all Digimarc Server needs. If you already have your Digimarc License directly with Digimarc, please call Equilibrium to purchase the Digimarc iKit that enables the automatic Digimarc processing to occur within a MediaRich MediaScript environment.

If no watermark is detected in the image, "undefined" is returned. If a watermark is detected, a dictionary (MediaScript object) is returned that contains key/value pairs that describe the embedded watermark. The particular pairs that are returned is a function of the watermark type.

For all watermark types, a 'Type' value is returned that indicates the type of the watermark. Possible values are: `basic, image, transaction, copyright`, and `unknown`.

If "unknown" is returned as the 'Type' value, no other data is returned. For the other four types, the following values are always returned: `CreatorID, Adult, Restricted`, and `CopyProtected`.

If "image" is returned as the 'Type' value, the following values are also returned: `DistributorID` and `ImageID`.

If "transaction" is returned as the 'Type' value, the following values are also returned: `DistributorID, DistributorPin`, and `TransactionID`.

If "copyright" is returned as the 'Type' value, the following values are also returned: `DistributorID, DistributorPin, Year1`, and `Year2`.

Syntax

```
digimarcRead();
```

Parameters

There are no parameters for this function.

Example

```
// Load an image
var image = new Media();
image.load(name @ "images:tif/32bit.tif");

// Check to see if there's a watermark in the image (there shouldn't be)
print("HasWM: " + (image.digimarcDetect()? "YES" : "NO") + "\n");

// Watermark the image
image.digimarcEmbed(CreatorID @ 404407, Adult @ 1, CopyProtected @ 1, Type @ "image",
DistributorID @ 0, ImageID @ 111);

// Now read the watermark back out, and print info about it
var info = image.digimarcRead();
print("Type = " + info.Type + "\n");
print("CreatorID = " + info.CreatorID + "\n");
print("Adult = " + info.Adult + "\n");
print("Restricted = " + info.Restricted + "\n");
print("CopyProtected = " + info.CopyProtected + "\n");
print("DistributorID = " + info.DistributorID + "\n");
print("ImageID = " + info.ImageID + "\n");
```

discard()

The `discard()` method 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()

The `drawText()` method composites the specified text string onto the image. This function fully supports CMYK image operations.

The foreground color can 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 colorspace), 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, rgb, or cmyk>]
[Index @ <value 0..16777215>]
[Unlock @ <color in hexadecimal, rgb, or cmyk>]
[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 *local.properties* file to change the default fonts directory. Refer to the *MediaRich Installation and Administration Guide* for more information.

MediaRich also allows you to set up virtual file systems and then use the `Font` parameter to specify fonts from that file system. Virtual file systems are defined in the MediaRich server’s local.properties file.

For example, if you define “CorpFonts:” to represent the path “C:/Fonts/CorpFonts/” in the local.properties file, you can use files from the CorpFonts directory with the `drawText()` function similar to the following:

```
image.drawText(Font @ "CorpFonts:/Arial", Text @ "Automated Imaging Solutions", Size
@ 18, Color @ 0x0000FF, x @ 185, y @ 30, Smooth @ true, Kern @ true);
```

`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 or Linux.

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

- `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()" on page 188.

This parameter supports a hexidecimal, RGB, or CMYK color specification:

- **hexidecimal** - color value expressed as a value from 0x000000 to 0xFFFFFF (RGB colorspace) or from 0x00000000 to 0xFFFFFFFF (CMYK colorspace)
- **RGB** - color value expressed as a value from 0 to 16,777,215
- **CMYK** - color value expressed as a value from 0 to 4,294,967,295

Colorspace

Always pass a color value appropriate to the colorspace. You can ensure this using the getPixelFormat() function in your script and then using different hexadecimal values for the RGB and the CMYK colorspaces in an IF/THEN construction. If getPixelFormat() returns "CMYK," use the CMYK value (0x plus eight more digits), and otherwise use the RGB value (0x plus six more digits).

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

Important: 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 an .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.

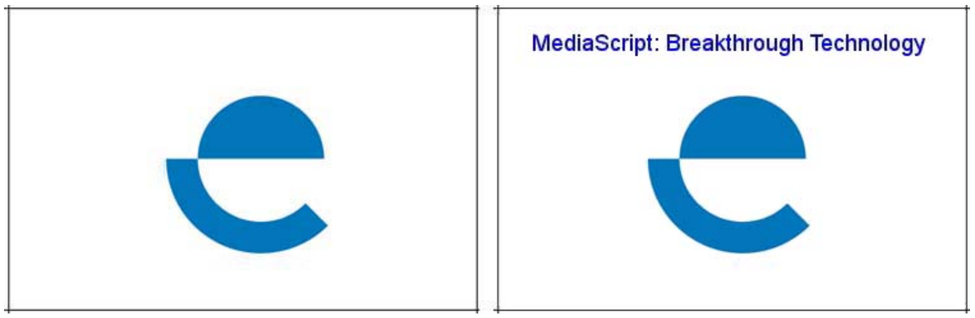
`ClearType` - if specified as `true`, the Windows ClearType text renderer will be used if available.

`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 or Linux.

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()

The `dropShadow()` method 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.

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()" on page 188.

Note: The `Enlarge` parameter is 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 137.

`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.makeText(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()

The `ellipse()` method 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 can 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 @ <true, false>]
[Color @ <color in hexadecimal, rgb, or cymk>]
[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. This parameter supports a hexidecimal, RGB, or CMYK color specification:

- **hexidecimal** - color value expressed as a value from 0x000000 to 0xFFFFFF (RGB colorspace) or from 0x00000000 to 0xFFFFFFFF (CMYK colorspace)
- **RGB** - color value expressed as a value from 0 to 16,777,215
- **CMYK** - color value expressed as a value from 0 to 4,294,967,295

Colorspace

Always pass a color value appropriate to the colorspace. You can ensure this using the getPixelFormat() function in your script and then using different hexadecimal values for the RGB and the CMYK colorspaces in an IF/THEN construction. If getPixelFormat() returns "CMYK," use the CMYK value (0x plus eight more digits), and otherwise use the RGB value (0x plus six more digits).

`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.

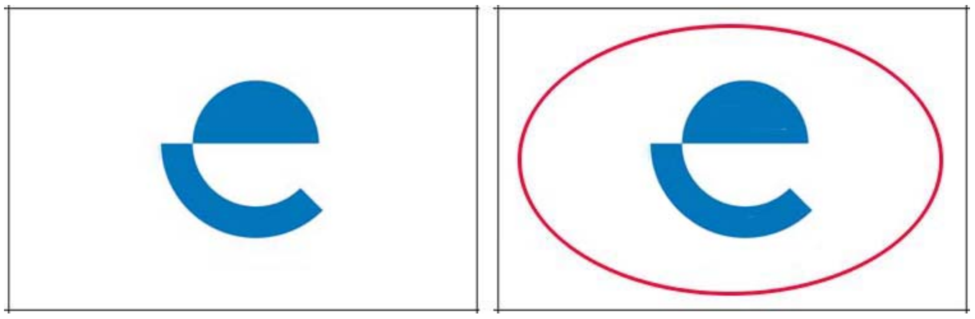
`Smooth` - if set to `true`, makes the edges of the ellipse smooth, preventing a pixellated effect. The default is `false`.

Important: If you are using smoothing for media that contains an alpha channel and you plan to save it to a format that does not support alpha channels, it is necessary to use convert() to remove the alpha channel before using this operation. Or, as an alternative, you can composite the modified image onto an opaque background before saving the image. Without this additional handling, the media will not look correct in a non-alpha file format.

`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()

The `embeddedProfile()` method 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()

The `equalize()` method 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”—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see “selection()” on page 186.

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()

The `exportChannel()` method exports a single channel of the source as a grayscale image. This function fully supports the CMYK colorspace.

Note: This function was formerly named `exportGun()`, which is 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 137.

Important: Unless a single layer is specified when exporting a single channel from a multi-layered image, the content of all layers are replaced with the specified channel. Because this strips away the transparency from the layers, only the topmost layer is visible when the image is collapsed or saved.

If you need the channel from a layer other than the top one, specify this single layer when loading the file, exporting the channel, or collapsing the image. Alternatively, you can use `Media.getLayer()` to retrieve the resulting layer from the image after using `exportChannel ()`.

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()

The `fixAlpha()` method 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.

Important: This function is now **removed** from MediaScript. If it is encountered, no operation is executed.

Syntax

```
fixAlpha();
```

Example

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

flip()

The `flip()` method flips the Media vertically or horizontally. This function fully supports images within the CMYK colorspace.

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 137.

Example

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

frameAdd()

The `frameAdd()` method 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 colorspace.

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. Refer to the *MediaRich Installation and Administration 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 is 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");
```

getAverageColor()

The `getAverageColor()` method returns the average color of the entire image as an integer that represents an RGB or CMYK color value.

Syntax

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

Parameters

This function has no parameters.

Example

```
var pix = new RgbColor(image.getAverageColor());
print("Average RGB color is "+pix.red+", "+pix.green+", "+pix.blue+"\n");
```

getBitsPerSample()

The `getBitsPerSample()` method returns the number of bits per sample.

Syntax

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

Parameters

This function has no parameters.

Example

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

getBytesPerPixel()

The `getBytesPerPixel()` method 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()

The `getFrame()` method 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()

The `getFrameCount()` method 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()

The `getHeight()` method returns the vertical size in pixels.

Syntax

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

Parameters

This function has no parameters.

Example

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

getImageFormat()

The `getImageFormat()` method returns a string representing the image type.

Important: `getImageFormat()` is now deprecated. `getPixelFormat()` is the preferred function and is supported in current versions of MediaScript (see "getPixelFormat()" on page 125).

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()

The `getInfo()` method returns the system version information. It is recommended for advanced users only.

Syntax

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

Parameters

This method takes one of these arguments as a string: `all`, `devices`, `commands`, or `filing`.

Example

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

getLayer()

The `getLayer()` method 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()

The `getLayerBlend()` method 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: The `Burn` mode is deprecated. `ColorBurn` results in the same blend.

Example

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

getLayerCount()

The `getLayerCount()` method 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()

The `getLayerEnabled()` method 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()

The `getLayerHandleX()` method returns the `HandleX` value (`left`, `center`, or `right`) of the Media layer with the specified index (if available). `HandleX` refers to the selected layer 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()

The `getLayerHandleY()` method returns the `HandleY` value (`top`, `middle`, or `bottom`) of the Media layer with the specified index (if available). `HandleY` refers to the selected layer 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()

The `getLayerIndex()` method 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()

The `getLayerName()` method 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()

The `getLayerOpacity()` method returns the opacity of the Media layer with the specified index (if available). For more information about opacity settings, see "composite()" on page 91.

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()

The `getLayerX()` method 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. See "collapse()" on page 84 for more information.

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()

The `getLayerY()` method 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. See "collapse()" on page 84 for more information.

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()

The `getMetadata()` method 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.

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()

The `getPalette()` method 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()

The `getPaletteSize()` method 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()

The `getPixel()` method 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 137.

Example

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

getPixelFormat()

The `getPixelFormat()` method returns a string representing the image type: `Gray-8`, `RGB-15`, `RGB-16`, `RGBA-16`, `RGB-24`, `RGBA-32`, `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()

The `getPixelTransparency()` method 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 colorspace.

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 137.

getPopularColor()

The `getPopularColor()` method returns the 24-bit color value (0 - 16,777,215) of the color that appears most frequently in the named object for the RGB colorspace.

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 137.

Example

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

getResHorizontal()

The `getResHorizontal()` method returns the horizontal resolution in DPI.

Syntax

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

Parameters

This function has no parameters.

Example

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

getResVertical()

The `getResVertical()` method returns the vertical resolution in DPI.

Syntax

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

Parameters

This function has no parameters.

Example

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

getSamplesPerPixel()

The `getSamplesPerPixel()` method returns the number of samples per pixel.

Syntax

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

Parameters

This function has no parameters.

Example

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

getWidth()

The `getWidth()` method returns the horizontal size in pixels.

Syntax

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

Parameters

This function has no parameters.

Example

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

getXmlInfo()

The `getXmlInfo()` method returns an XML document that contains the installed file formats. This document looks similar to the 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()

The `glow()` method 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.

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 colorspace (see "setColor()" on page 188).

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 colorspace), 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 colorspace), 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 `0`.

`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 188.

`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 137.

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()

The `gradient()` method composites a color gradient onto the source image. This method accepts all `composite()` parameters except `HandleX` and `HandleY`. For information about these parameters, see "composite()" on page 91.

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 @ <0..255>]
[blend @ <"Normal", "Darken", "Lighten", "Hue", "Saturation", "Color", "Luminosity",
"Multiply", "Screen", "Dissolve", "Overlay", "HardLight", "SoftLight", "Difference",
"Exclusion", "Dodge", "ColorBurn", "Under", "Colorize", "Prenormal">]
[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. The `adjust` parameter should be set to false if one is creating an image consisting of nothing but a gradient.

`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 137.

Example

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

importChannel()

The `importChannel()` method 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 is now deprecated.

Syntax

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

Parameters

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

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

> ***Note:*** The `Name` parameter is now deprecated.

`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 137.

`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`.

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()

The `infoText()` method 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 102.

`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 or Linux.

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()

The `line()` method draws a line across the image based on the specified parameters. This method accepts all `composite()` parameters except `HandleX` and `HandleY`. For information about these parameters, see "composite()" on page 91.

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. 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 @ <true, false>]
[Color @ <color in hexadecimal, rgb, or cymk>]
[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 to be used for the line. The default is the foreground color. This parameter supports a hexidecimal, RGB, or CMYK color specification:

- **hexidecimal** - color value expressed as a value from 0x000000 to 0xFFFFFF (RGB colorspace) or from 0x00000000 to 0xFFFFFFFF (CMYK colorspace)
- **RGB** - color value expressed as a value from 0 to 16,777,215
- **CMYK** - color value expressed as a value from 0 to 4,294,967,295

Colorspace

Always pass a color value appropriate to the colorspace. You can ensure this using the getPixelFormat() function in your script and then using different hexadecimal values for the RGB and the CMYK colorspaces in an IF/THEN construction. If getPixelFormat() returns "CMYK," use the CMYK value (0x plus eight more digits), and otherwise use the RGB value (0x plus six more digits).

`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`.

> ***Note:*** The `Burn` option is now deprecated; `ColorBurn` results in the same blend.

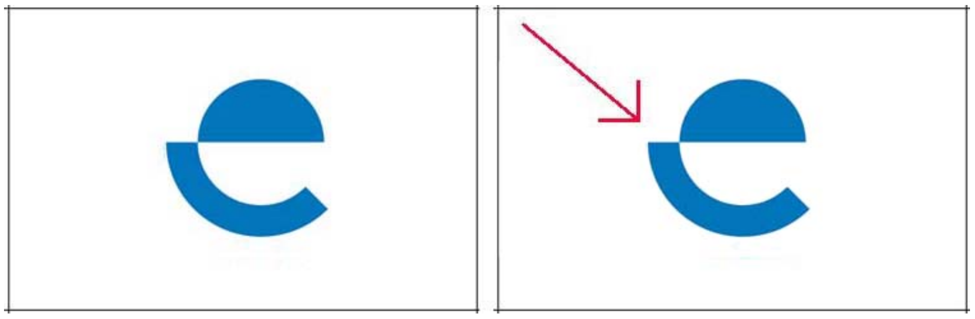
`Width` - specifies the thickness (in pixels) of the line. The default is `1`.

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

> ***Important:*** If you are using smoothing for media that contains an alpha channel and you plan to save it to a format that does not support alpha channels, it is necessary to use convert() to remove the alpha channel before using this operation. Or, as an alternative, you can composite the modified image onto an opaque background before saving the image. Without this additional handling, the media will not look correct in a non-alpha file format.

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()

The `load()` method loads an image into the Media object from the specified file. For a detailed list of file format load (read) support, see "File Format Support" on page 364.

Note: In MediaRich version 3.6 and later, `load()` does not perform any color conversion. For instance, additional parameters for Color Profile Specifications `srcProfile`, `destProfile`, and `intent` are not supported. You must explicitly call `convert()` or `colorConvert()` to change an image type.

The following file formats support file sizes greater than 4GB: .tif (BigTIFF), .psb, and .pdf.

Important: Loading, modifying, and saving very large image files can result in errors or crashes when the system cannot accommodate these files. For more information, see "Memory Issues with Very Large Image Files" on page 362.

Also, when working with large images, ensure that your page file size is set to "System Managed Size" on the enabled drive and make sure that the drive has enough space to contain it.

The following file formats support the CMYK colorspace: .ai, .eps, .pdf, .psd, .tif, and .jpg.

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>]
[LoadMetadata @ <true, false>]
[frames @ <"frames list">] // TIFF, GIF, PS, PDF, INDD files only
[layers @ <"layer list">] // PSD files only
[collapsed @ <true, false>] // PSD files only
[VisibleOnly @ <true, false>] // PSD files only
[PreviewAlpha @ <true, false>] // 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>] // AI, EPS, PDF, and PS files only
[useCMYK @ <true, false>] // AI, EPS, PDF, and PS files only
[intermediateFileName @ <"filePath">] // AI, EPS, PDF, and PS files only
[IgnoreHeader @ <true, false>] // AI, EPS, PDF, and PS files only
[MaxWidth @ <integer value>] // TIFF, JPEG, and PS files only
[MaxHeight @ <integer value>] // TIFF, JPEG, and PS files only
[AllowMangled @ <true, false>] // PNG only
[resolution @ <integer value>] // multi-resolution TIFF
```

```
[time @ <seconds>] // SWF files only
[Xs @ <size>] // SWF files only
[Ys @ <size>] // SWF files only
[InterpolationQuality @ <"off"><"auto"><"linear"><"vng"><"ahd"> // Raw Camera only
[WhiteBalance @ <"auto"><"camera">] // Raw Camera only
[AdjustWhiteBalance @ "v1, v2, v3, v4"] // Raw Camera only
[ColorSpace @ <"raw"><"srgb"><"adobe"><"wide"><"prophoto"><"xyz">] // Raw
[NoFujiRotate @ <true><false>] // Raw Camera only
[UseFujiSecondary @ <true><false>] // Raw Camera only
[BadPixelFile @ "<filename>"] // Raw Camera only
[ExternalJPEGFile @ "<filename>"] // Raw Camera only
[FourColorInterpolate @ <true><false>] // Raw Camera only
[HighlightMode @ <0..9>, default = 0] // Raw Camera only
[BlackPoint @ <integer value>] // Raw Camera only
[Brightness @ <0..1>, default = 1.0] // Raw Camera only
[BilateralFilterDomain @ <floating point value>] // Raw Camera only
[BilateralFilterRange @ <floating point value>] // Raw Camera only
[HalfSize @ <true, false >] // Raw Camera only
[Thumbnail @ <true, false >] // Raw Camera only
[Brighten @ <true, false >] // Raw Camera only
[UseEmbeddedMatrix @ <true, false >] // Raw Camera only
[Select @ <number>] // Raw Camera only
[NoStretchRotateRaw] @ <true, false > // Raw Camera only
[DocMode @ <true, false >] // Raw Camera only
[DocModeNoScaling @ <true, false >] // Raw Camera only
[NoiseThreshold @ <number>] // Raw Camera only
[FixContrast @ <true, false>] // Raw Camera only
[PassThrough @ <string>] // Raw Camera only
[Dpi @ <value 1..32767>] // Office only
[Pages @ <pages>] // Office only
[Brightness @ <brightness>] // Office only
[Dither @ <dither>] // Office only
[ImageWidth @ <width>] // Office only
[ImageHeight @ <height>] // Office only
[Grayscale @ <true, false>] // Office only
[Scale @ <scale>] // Office only
);
```

Note: DPI is supported for non-bitmap (vector) file formats such as AI, EPS, PDF, PS and all Office files in the LibreOffice environment. It has **no** effect on the loading of bitmap (raster) images such as BMP, Targa, TIFF, GIF, and JPEG.

Basic Parameters

Use the following parameters for basic load() functionality.

Parameter	Usage
`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: `MediaRichCore/Shared/Originals/Media`. Refer to the *MediaRich CORE Installation and Administration Guide* for more information about modifying this default directory. You can also load a file from an HTTP or FTP URL using the this parameter. **Note:** The functionality of loading files from HTTP or FTP sources is enabled by default. If you need to disable this for security reasons, contact your MediaRich administrator.
`type`	Specifies the expected file type When this parameter is not specified, the type is derived from the file extension. Valid type names are: bmp, eps, 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 Web site 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. **Note:** Some image formats do not support the this parameter. If the filename extension is not a recognized type in such a case, the load method returns a "File type wrong" error. Some formats in this list include PDF, AI, EPS, EPSF, PS, and all of the formats handled by LibreOffice.
`LoadMetadata`	When `true`, loads any Exif, IPTC, or XMP metadata associated with the image The default is value `false`. For more information about metadata support, see "MediaRich Metadata Support" on page 326.

Loading PNG Files

The `load()` method supports additional parameters used for loading .png files.

Parameter	Usage
`fillalpha`	When set to `true`, fills transparent and translucent pixels with the image 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	Usage
`AllowMangled`	When set to `true`, allows a mangled PNG image to be partially read into the Media object The image might be distorted, but you can salvage as much of it as possible. The default is `false`. This is supported to correct problems with other png software and reports an error and aborts if the image cannot be fully read successfully.

Loading JPEG Files

The `load()` method supports the following additional parameters used for loading .jpeg and .jpg files.

Parameter	Usage
`MaxWidth` `MaxHeight`	Set the maximum width and height for the JPEG image The file is not deleted after the operation completes. For more information, see "Loading Files with MaxHeight and MaxWidth" on page 153.

Loading TIFF Files

The `load()` method supports the following additional parameters used for loading .tif and .tiff files.

Parameter	Usage
`resolution`	Specifies a value used to read a specified resolution from a multi-resolution TIFF image The value is an integer ranging from 0 to the number of resolutions - 1. The highest resolution (the largest image) is 0. Each subsequent resolution gives an image of half the dimension of the previous. For example, a resolution of 2 results in an image with width and height that is 1/4 of the original. A resolution of 4 results in an image wiht width and height that is 1/16 the original. If the resolution does not exist, a NoMedia error is returned.
`MaxWidth` `MaxHeight`	Set the maximum width and height for the JPEG image The file is not deleted after the operation completes. For more information, see "Loading Files with MaxHeight and MaxWidth" on page 153.

Loading Raw Camera Files

Note: Some of the extensions on Raw Camera files can conflict with other file types and cause issues with loading. For more information about addressing this issue, see "Loading Raw

Camera Files” on page 361.

The `load()` method supports the following additional parameters used for loading Raw Camera files.

Parameter	Usage
`InterpolationQuality`	Sets the interpolation quality mode Defaults to `ahd`, which is the slowest but highest quality. Controls how the color pixels from the CCD are interpolated from a non-linear color pattern to a single RGB pixel. `linear` uses a linear algorithm; this is the fastest but the lowest quality. `vng` forces a threshold-based variable number of gradients algorithm to be used. `ahd` forces an adaptive homogeneity-directed algorithm to be used, and is usually the default when `auto` is specified. `off`, no interpolation is done and you get a grayscale image with the CCD pixel data just as it came from the camera. Controls the -q param in dcraw.
`WhiteBalance`	Specifies the white balance value The default is `auto`, but specify `camera` to use the white balance values (if available) for the camera.
`AdjustWhiteBalance`	Specifies multipliers to adjust the image white balance manually This overrides the `WhiteBalance` parameter. The values are floating point numbers. If you specify `0` for the first value, there is no manual white balance applied.
`ColorSpace`	Specifies the colorspace to use when converting the image from raw data to an actual image. Default is `srgb`. `raw` causes colorspace conversion to be disabled.
`NoFujiRotate`	When set to `true`, disables the default rotation of Fuji Super CCD images Fuji Super CCD images are natively stored at a 45 degree angle, which is why the default is to rotate these images so they look normal.
`BadPixelFle`	Specifies the path to the "bad pixel file" to use for image correction If you have “bad pixel file” that indicates which CCD elements are defective in your camera, you can specify the full path to that file and the interpolation code will try and work around those bad pixels. By default it will not exclude any pixels.

Parameter	Usage
`ExternalJPEGFile`	Specifies the path for external image information file Some cameras only save raw images as a “debug” mode feature, and that data sometimes does not contain all of the information you want to have associated with the image. If you specify the path to this file with this parameter and it is a camera that supports this external information, it loads that information as part of the image. By default it does not look for external files.
`FourColorInterpolate`	When set to `true`, interpolates RGGB as four colors instead of three Defaults to `false`.
`HighlightMode`	Specifies the highlight recovery mode for clipped highlights `0` (the default) omits any recovery, `1` specifies clip channel data, `2` through `9` specify the attempt to recover the clipped channel data.
`BlackPoint`	Specifies the black point to use, instead of the one specified in the image data Default is to use image value.
`Brighten`	When set to `true`, automatically brightens the image This is equivalent to the -w param in dcraw. Defaults to `false`.
`Brightness`	Specifies the brightness setting for the image Defaults to 1.72 when "Brighten" is disabled or 1.00 if "Brighten" is enabled. Lower values darken the overall image. Valid values range from 0.0 to 100.0.
`BilateralFilterDomain` `BilateralFilterRange`	Specify the bilateral filter domain and range When specified, a bilateral filter is applied to the image data to reduce the noise present in CCDs. By default no filtering is done. **Important:** If you specify either of these values, you must specify both or a missing parameter error results.
`HalfSize`	When set to `true`, loads the image at half the width and height for greater speed This is equivalent to the -h param in dcraw. Defaults to `false`.
`Thumbnail`	When set to `true`, loads the image from the embedded thumbnail, if present This is equivalent to the -e param in dcraw. Defaults to `true`.

Parameter	Usage
`UseEmbeddedMatrix`	When set to `true`, uses the embedded color matrix in the file to adjust the colors of the image This option only affects Olympus, Leaf, and Phase 1 raw camera images. Defaults to `true`.
`Select`	Controls which image is returned from raw files that can contain multiple images This is equivalent to the -s param in dcraw. Defaults to `0`.
`NoStretchRotateRaw`	When set to `true`, equivalent to the -j param in dcraw. Defaults to `false`.
`DocMode`	When set to `true`, equivalent to the -d param in dcraw. Defaults to `false`.
`NoiseThreshold`	When set to `true`, causes a noise reduction filter to be applied to the image This is equivalent to the -n param in dcraw. Defaults to `false`.
`FixContrast`	When set to `true`, helps to adjust the images to better match that of other software Defaults to `true`. **Note:** This parameter is ignored when the image comes from the thumbnail, as the image should technically already be auto-adjusted.
`PassThrough`	Passes command line string as options straight through to dcraw You can use this for additional control of dcraw for any parameters that are not supported via `load()` **Note:** Specifying some direct dcraw parameters can cause the image to fail to load, such as redirecting it to a file instead of standard output like this reader expects.

Loading AI, PS, EPS, and PDF Files

The `load()` method supports additional parameters used for loading .ai, .ps, .eps, and .pdf files.

Important: Illustrator (.ai) files can be rasterized only when the PDF information is embedded. In CS3 and earlier releases, the Illustrator default behavior was to save .ai files with PDF information. With the release of CS4, the “Create PDF compatible File” option is disabled by default for a Save As operation. All sample files included with CS4 do not have the embedded PDF information.

Parameter	Usage
`dpi`	Sets the resolution (Dots Per Inch) at which data is rendered The default is `150`, the current value of "`tiff_dpi`", specified in the ps2xxx.ini file. If `tiff_dpi` is not specified, the default is 75. If you use Ghostscript, the default is `72`. **Note:** You can modify this parameter to specify a lower DPI if you encounter the `FlySDK failed to render the file` error when loading any of these file types. **Important:** DPI is supported for non-bitmap (vector) file formats such as AI, EPS, PDF, PS and all Office files in the LibreOffice environment. It has **no** effect on the loading of bitmap (raster) images such as BMP, Targa, TIFF, GIF, and JPEG.
`useCMYK`	When set to `true`, loads any CMYK file as CMYK instead of converting to RGB The default value for this parameter is `false`, so all CMYK PS, EPS, and PDF files are converted to RGB by default.
`intermediateFileName`	Specifies the intermediate TIFF file used to render the file The file is not deleted after the operation completes. Instead of rendering pages into the source Media, you 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. Pages can still be loaded into the source Media as well.
`IgnoreHeader`	When set to true (default), causes the reader to not use the information in the EPS header line in the file.

Photoshop-specific parameters

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.

Parameter	Usage
`collapsed`	When set to `false`, overrides the single raster default loading and instead loads all layers
`layers`	Specifies the layers to load and the order in which to load them The layer numbers begin at 0 (background). To specify all layers (including non-visible layers) use the wild-card notation `*` inside of quotes. The `visibleOnly()` function can 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.
`VisibleOnly`	When set to `true` and loading photoshop layers (`collapsed` set to `false`), loads only the layers designated as visible The default is `false`.
`PreviewAlpha`	When set to `false` and layers are not specified, does not load the preview alpha channel in the image preview PreviewAlpha defaults to `0`, except when there is no special background layer in the PSD/PSB file and individual layers are not loaded or collapsed is set to false. When those conditions are met, it defaults to `1` (load the first preview image alpha channel). If this parmeter is explicitly specified, the setting overrides any of the defaults and can load any available preview image alpha if that alpha channel is present in the file. With the default (`0`), no preview alpha is loaded.
`MaxWidth` `MaxHeight`	Set the maximum width and height for the Photoshop image The file is not deleted after the operation completes. For more information, see "Loading Files with MaxHeight and MaxWidth" on page 153.

When you specify the `layers` parameter, 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 `*` as a wildcard when specifying layers. For example:

```
image.load(name @ "horizontal.psd", layers @ "B*")
```

Note: Layers of grayscale Photoshop files are not supported.

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 load 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.

Loading Vector Files Using Ghostscript

If the native rendering of vector images like .pdf or .eps does not meet all of your requirements, you can use Ghostscript as an alternative. Using Ghostscript requires installation on the same machine as the MediaRich Server and adding the Ghostscript .bin path to the local.properties file. For more information, refer to the *MediaRich CORE Installation and Administration Guide*.

The `load()` method supports the following parameters for loading vector files via Ghostscript.

```
[Adjoin]
[AntiAlias]
[Dpi=<integer 1..32767>]
[IntermediateFileName=<text>]
[gsopts @ "-dEPSCrop"]
[NoClip]
[Pages=<text>]
[UseCIEColor]
[UseCMYK]
[usecropbox]
[showGSCommand]
```

Note: The `Adjoin`, `AntiAlias`, `gsopts`, `NoClip`, `UseCIEColor`, `usecropbox`, and `showGSCommand` parameters work with a Ghostscript-enabled installation, not with the default MediaRich native vector image support.

Loading Microsoft Office Files

The complete LibreOffice 6.0 must be installed to enable the Office and other file type support in MediaRich.

For more information about installing LibreOffice and enabling support for Microsoft Office files, refer to the *MediaRich CORE Installation and Administration Guide*.

If the default LibreOffice environment does not meet your requirements for processing Office files, you can disable or not install LibreOffice and install the BlackIce drivers on the Windows platform. For more information about installing and configuring these drivers, , refer to the *MediaRich CORE Installation and Administration Guide*.

PowerPoint documents (*.ppt or *.ppsx)

> ***Note:*** Advanced formatting of PowerPoint documents is handled by LibreOffice or the BlackIce drivers when loading the .ppt or .ppsx file. This formatting might not match the pagination applied by the Microsoft Office application.

For the default LibreOffice environment:

```
load(Dpi @ <value 1..32767>, Pages @ <pages>);
```

For the BlackIce/Microsoft Office environment:

```
load(ImageWidth @ <width>, ImageHeight @ <height>, Pages @ <pages>, GrayScale @
<true, false>);
```

Loads a PowerPoint file into a multi-frame media object. Individual slides can be accessed with the getFrame() method.

```
Dpi @ <value 1..32767>
```

DPI controls the size that images come are loaded. When `Dpi` is declared for an Office file, MediaRich will rasterize the source data at a smaller or larger pixel-size in relation with the smaller or larger amount declared. If `Dpi` is undeclared, source data coming in from these mechanisms will be rasterized at 150 DPI.

> ***Note:*** DPI is supported for non-bitmap (vector) file formats such as AI, EPS, PDF, PS and all Office files in the LibreOffice environment. It has **no** effect on the loading of bitmap (raster) images such as BMP, Targa, TIFF, GIF, and JPEG.

```
ImageWidth @ <width>
```

Optional integer parameter to define the output width in pixels. Default is the slide width in the presentation.

> ***Note:*** ImageWidth is supported in the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

```
ImageHeight @ <height>
```

Optional integer parameter to define the output height in pixels. Default is the slide height in the presentation.

> ***Note:*** ImageHeight is for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

```
Pages @ <pages>
```

Optional integer parameter to define the slides to be imaged. Default is all slides.

```
Grayscale @ <true, false>
```

Optional Boolean parameter to specify grayscale. Default is false.

Note: Greyscale is supported for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

```
Brightness @ <brightness>
```

Optional integer parameter to specify image brightness. Only affects grayscale images. Range is 0-199. Default is 100.

Note: Brightness is supported for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

```
Dither @ <dither>
```

Optional enumerated parameter. Possible values are: "None", "Floyd-Steinberg", "Jarvis-Judice-Ninke", "Smooth", "Sharp", "Stucki", and "Threshold" representing the various dithering algorithms supported by the driver. Only affects grayscale images. Default is "Floyd-Steinberg".

Note: Dither is supported for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

A PowerPoint document is always presented in the original's aspect ratio of width to height. If both height and width are specified and are not in the same ratio as the original document's, the resulting image will be padded in white to conform to the requested height and width.

```
IntermediateFileName @ "<pathname>"
```

This parameter takes a pathname of the PDF file to generate instead of loading the data into the media object. This parameter also works on OpenOffice - Write, Draw, Calc and Impress Documents.

Word Documents (*.doc)

Note: Pagination and advanced formatting of Word documents is handled by LibreOffice or the BlackIce drivers when loading the .doc file. This pagination and formatting might not match the pagination applied by the Microsoft Office application.

For the default LibreOffice environment:

```
load(Dpi @ <dpi>, Pages @ <pages>);
```

For the BlackIce/Microsoft Office environment:

```
load(ImageWidth @ <width>, ImageHeight @ <height>, Pages @ <pages>, GrayScale @
<true, false>);
```

Loads a Microsoft Word document into a multi-frame media object. Individual pages can be accessed with the getFrame method.

```
Dpi @ <value 1..32767>
```

DPI controls the size that images come in. When Dpi is declared, MediaRich will rasterize the source data at a smaller or larger pixel-size in relation with the smaller or larger amount declared. If Dpi is undeclared, source data coming in from these mechanisms will be rasterized at 150 DPI.

Note: DPI is supported for non-bitmap (vector) file formats such as AI, EPS, PDF, PS and all Office files in the LibreOffice environment. It has **no** effect on the loading of bitmap (raster) images such as BMP, Targa, TIFF, GIF, and JPEG.

`ImageWidth @ <width>`

Optional integer parameter to define the output width in pixels. Default is the page width in pixels as if printed at 72 DPI.

Note: ImageWidth is supported for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

`ImageHeight @ <height>`

Optional integer parameter to define the output height in pixels. Default is the page height in pixels as if printed at 72 DPI.

Note: ImageHeight is supported for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

`Pages @ <pages>`

Optional integer parameter to define the pages to be imaged. Default is all pages.

`Grayscale @ <true, false>`

Optional Boolean parameter to specify grayscale. Default is false.

Note: Greyscale is supported for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

`Brightness @ <brightness>`

Optional integer parameter to specify image brightness. Only affects grayscale images. Range is 0-199. Default is 100.

Note: Brightness is supported for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

`Dither @ <dither>`

Optional enumerated parameter. Possible values are: "None", "Floyd-Steinberg", "Jarvis-Judice-Ninke", "Smooth", "Sharp", "Stucki", and "Threshold" representing the various dithering algorithms supported by the driver. Only affects grayscale images. Default is "Floyd-Steinberg".

Note: Dither is supported for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

For Word documents, either the ImageWidth can be specified or the ImageHeight, but not both. The aspect ratio of the document is always preserved.

```
IntermediateFileName @ "<pathname>"
```

This parameter takes a pathname of the PDF file to generate instead of loading the data into the media object. This parameter also works on OpenOffice - Write, Draw, Calc and Impress Documents.

Excel documents (*.xls)

Note: Pagination and advanced formatting of Excel documents is handled by LibreOffice or the BlackIce drivers when loading the .xls file. This pagination and formatting might not match the pagination applied by the Microsoft Excel application.

For the default LibreOffice environment:

```
load(Dpi @ <value 1..32767>, Pages @ <pages>);
```

For the BlackIce/Microsoft Office environment:

```
load(ImageWidth @ <width>, ImageHeight @ <height>, Pages @ <pages>, GrayScale @
<true, false>, Scale @ scale);
```

Loads a Microsoft Excel document into a multi-frame media object. Individual worksheets can be accessed with the getFrame method.

```
Dpi @ <value 1..32767>
```

DPI controls the size that images come in. When Dpi is declared, MediaRich will rasterize the source data at a smaller or larger pixel-size in relation with the smaller or larger amount declared. If Dpi is undeclared, source data coming in from these mechanisms is rasterized at 150 DPI.

Note: DPI is supported for non-bitmap (vector) file formats such as AI, EPS, PDF, PS and all Office files in the LibreOffice environment. It has **no** effect on the loading of bitmap (raster) images such as BMP, Targa, TIFF, GIF, and JPEG.

```
ImageWidth @ <width>
```

Required integer parameter to define the output width in pixels.

Note: ImageWidth is supported for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

```
ImageHeight @ <height>
```

Required integer parameter to define the output height in pixels.

Note: ImageHeight is supported for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

```
Pages @ <pages>
```

Optional integer parameter to define the worksheets to be imaged. Default is all worksheets.

```
Grayscale @ <true, false>
```

Optional Boolean parameter to specify grayscale. Default is false.

> ***Note:*** Greyscale is supported for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

`Scale @ <scale>`

The scale factor for the worksheet image, ranging from 0.000000 to 1.000000.

> ***Note:*** Scale is supported for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

`Brightness @ <brightness>`

Optional integer parameter to specify image brightness. Only affects grayscale images. Range is 0-199. Default is 100.

> ***Note:*** Brightness is supported for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

`Dither @ <dither>`

Optional enumerated parameter. Possible values are: "None", "Floyd-Steinberg", "Jarvis-Judice-Ninke", "Smooth", "Sharp", "Stucki", and "Threshold" representing the various dithering algorithms supported by the driver. Only affects grayscale images. Default is "Floyd-Steinberg".

> ***Note:*** Dither is supported for the BlackIce/Microsoft Office environment and does not apply to rasterizing such documents using the default LibreOffice mechanism.

For Excel documents, that portion of the worksheet that fits in the specified width and height is imaged. Use the scale parameter to control how much of the worksheet appears.

`IntermediateFileName @ "<pathname>"`

This parameter takes a pathname of the PDF file to generate instead of loading the data into the media object. This parameter also works on OpenOffice - Write, Draw, Calc and Impress Documents.

In addition, only worksheets are imaged. Charts are only imaged if embedded within a worksheet.

Also, a default installation of Microsoft Excel will not allow an .xls document containing macros to open without user intervention. If there is a possibility of this type of document being processed, the default settings in Excel must be changed. In that program, use Tools > Macro > Security to make the necessary changes.

Loading WBMP Files

The `load()` method supports the following additional parameter used for loading .wbmp files.

Parameter	Usage
`waplook`	When set to `true`, sets the image palette to simulate the look of an LCD screen on an actual WAP device

Loading SWF Files

SWF reading works like other multpage/multiframe readers. You can specify the frame or frames you want to load into the media object with the frame or frames parameter used with `Media.load()`. Ranges such as "4-10" for example are also valid as are specific frame numbers, such as "4,8,20".

You can then operate on the frames as you would with any multipage/multiframe media object, using `Media.getFrame()`.

> ***Note:*** Extracting multiple frames causes a temp image file to be created for each frame on disk, so do not try to convert entire movies this way since it will take a very long time and may cause disk full errors.
>
> Also, because it is NOT possible to seek in SWF files, if you specify frames deep into the movie the extraction can take a very long time because the movie must be played back sequentially internally.

The frames have no times present—if you want to convert a short sequence of frames into a GIF animation, you must specify the frame rate during the save() operation.

The `load()` method support the following additional parameters used for loading .swf files.

Parameter	Usage
`time`	Specifies the number of seconds into the movie to return a single frame This parameter always returns a frame from readable movies. Requesting a specific frame could fail if it is beyond the number of frames in the movie or past user input sections.
`Xs` `Ys`	Force the frame to be rendered at a specified size If these parameters are not specified, the default size stored in the SWF file is used.

SWF Examples

```
var m = new Media();
m.load(name @ "Example.swf", time @ 10);
m.save(name @ "out.tif");

m.load(name @ "Example.swf", frame @ 45);
m.save(name @ "out.jpg");

m.load(name @ "Example.swf", frames @ "0,10,20");
m.save(name @ "out.tif");

m.load(name @ "Example.swf", frames @ "7-12");
m.save(name @ "out.tif");
```

Loading Multiple-Page/Frame Files

The `load()` method supports the `frames` parameters for loading multiple-page/frame files (.tif, .gif, .ps, .pdf, .eps, and .indd) files.

By default, all pages/frames are loaded. To load specific pages, specify a page range. `frame` and `page` are valid aliases for `frames`. Any of the following are valid:

- `frames “3-9”`
- `frames 7`
- `frame 12`
- `frame “4-9”`

Note: Alpha channels in .ps/.pdf/.eps documents are only preserved when a single page is specified

Loading Files with MaxHeight and MaxWidth

Instead of rendering pages into the source Media, you can use the `load()` method 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. Pages can still be loaded into the source Media as well.

The image dimensions are constrained so as to keep the aspect ratio correct. The constraining dimension is whichever one stored in the image passes the maximum set, or if both do, whichever one has the largest value.

Important: If not specified, the `MaxWidth` and `MaxHeight` parameters default to the values defined in the properties file `global.Media.Load.MaxWidth` and `global.Media.Load.MaxHeight`, or 8192 for each if they are not defined anywhere. A value of 0 (zero) disables downscaling so images are loaded at full resolution. These parameters work independently, and scaling is always proportional. So if, for example, width is specified as 0 and the height is specified as 4000, the image will be reduced proportionally as necessary so that the height is not larger than 4000.

Any image that is larger than these limits will be automatically scaled down as te image is loaded, pixel row by pixel row, so as to greatly reduce the memory footprint of large images and make the loading of extremely large images possible.

For more information about setting MediaLoadMaxHeight and MediaLoadMaxWidth to manage automatic scale-down, refer to the *MediaRich CORE Server Installation and Administration Guide* .

loadAsRgb()

The `loadAsRgb()` method 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 are converted to RGB using any embedded ICC profile and the default RBGB profile. If there is no ICC profile embedded in the file, the default CMYK profile is used. This function is defined in *Sys/media.ms*. See “MediaRich Color Management” on page 318 for more information.

Syntax

```
loadAsRgb(
[name @ <"filename">]
[type @ <"typename">]
[detect @ <true, false>]
[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 137.

Note: DPI is supported for non-bitmap (vector) file formats such as AI, EPS, PDF, PS and all Office files in the LibreOffice environment. It has **no** effect on the loading of bitmap (raster) images such as BMP, Targa, TIFF, GIF, and JPEG.

`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()

The `makeCanvas()` method creates a “blank” Media object of the specified dimensions and fully supports the CMYK colorspace.

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`. The default bit-depth is `RGBA-32` (RGB, 32-bit).

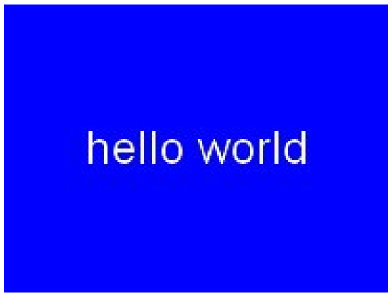
Note: The 16-bit type is 5-6-5, while the 16a-bit is 1-5-5-5 with the top bit as an alpha channel.

`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. This 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()

The `makeText()` method, 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 colorspace.

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>] //(win82ows 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 *local.properties* file to change the default fonts directory. Refer to the *MediaRich Installation and Administration 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 or Linux.

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 188.

`Rtype` - specifies the target bit depth. Supported bit-depths are: `Gray-8`, `RGB-15`, `RGB-16`, `RGBA-16`, `RGB-24`, `RGBA-32`, `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
- CMYK -> CMYK-32
- CMYKA -> CMYKA-40

Note: Deprecated `Rtype` values include: `Grayscale`, `pal-8`, `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 an .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 or Linux.

`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()

The `measureText()` Method 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.

Important: 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 *local.properties* file to change the default fonts directory. Refer to the *MediaRich Installation and Administration 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.

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`. 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 is 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 an .afm extension. This file must be present in order for kerning to be applied to the text.

noiseAddNoise()

The `noiseAddNoise()` Method applies random pixels to an image to simulate a noise effect.

Note: This function is "selection aware"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

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()

The `otherHighPass()` method 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"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

Syntax

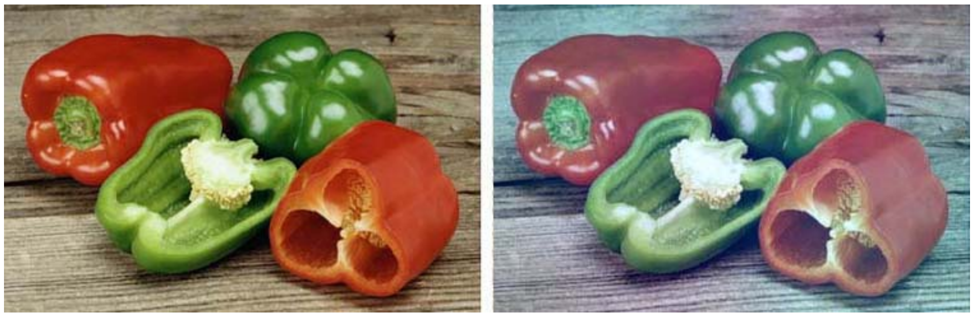
```
otherHighPass(
[Radius @ <value, 10..250>]
);
```

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()

The `otherMaximum()` method 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"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

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()

The `otherMinimum()` method 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"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

Syntax

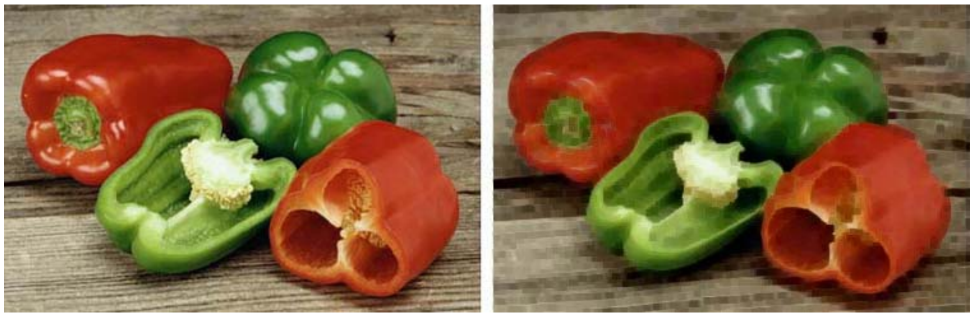
```
otherMinimum(
[Radius @ <value 1..10>]
);
```

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()

The `pixellateFragment()` method makes and offsets four copies of the image.

> ***Note:*** This function is "selection aware"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

Syntax

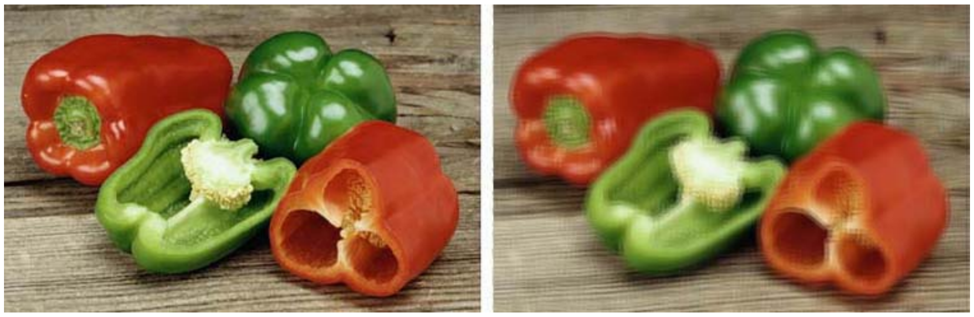
```
pixellateFragment(
[Radius @ <value 1..16>]
);
```

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()

The `pixellateMosaic()` method pixellates the image, with pixel size determined by the `Radius` parameter.

> ***Note:*** This function is "selection aware"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

Syntax

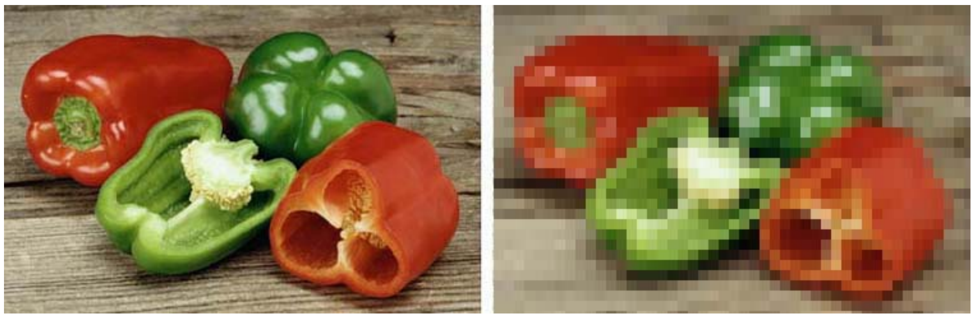
```
pixellateMosaic(
[Size <2..64>]
);
```

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()

The `polygon()` Method draws and positions a polygon on the image based on the specified parameters. This method accepts all composite() parameters except `HandleX` and `HandleY`. For information about these parameters, see “composite()” on page 91.

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. 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 @ <true, false>
[Color @ <color in hexadecimal, rgb, or cymk>]
[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. The default is the foreground color. This parameter supports a hexidecimal, RGB, or CMYK color specification:

- **hexidecimal** - color value expressed as a value from 0x000000 to 0xFFFFFF (RGB colorspace) or from 0x00000000 to 0xFFFFFFFF (CMYK colorspace)
- **RGB** - color value expressed as a value from 0 to 16,777,215
- **CMYK** - color value expressed as a value from 0 to 4,294,967,295

Colorspace

Always pass a color value appropriate to the colorspace. You can ensure this using the getPixelFormat() function in your script and then using different hexadecimal values for the RGB and the CMYK colorspaces in an IF/THEN construction. If getPixelFormat() returns "CMYK," use the CMYK value (0x plus eight more digits), and otherwise use the RGB value (0x plus six more digits).

`Index` - colorizes the polygon 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 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: The `Burn` option is 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.

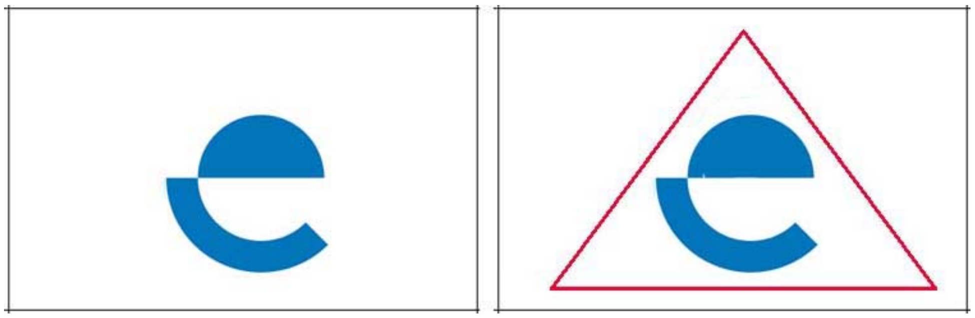
`Smooth` - if set to `true`, makes the edges of the polygon smooth, preventing a pixellated effect. The default is `false`.

Important: If you are using smoothing for media that contains an alpha channel and you plan to save it to a format that does not support alpha channels, it is necessary to use convert() to remove the alpha channel before using this operation. Or, as an alternative, you can composite the modified image onto an opaque background before saving the image. Without this additional handling, the media will not look correct in a non-alpha file format.

`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()

The `quadWarp()` method 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 colorspace.

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 137.

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()

The `rectangle()` method draws and positions a rectangle on the image based on the specified parameters. This method accepts all `composite()` parameters except `HandleX` and `HandleY`. For information about these parameters, see "composite()" on page 91.

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. 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 @ <true, false>]
[Color @ <color in hexadecimal, rgb, or cymk>]
[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. The default is the foreground color. This parameter supports a hexidecimal, RGB, or CMYK color specification:

- **hexidecimal** - color value expressed as a value from 0x000000 to 0xFFFFFF (RGB colorspace) or from 0x00000000 to 0xFFFFFFFF (CMYK colorspace)
- **RGB** - color value expressed as a value from 0 to 16,777,215
- **CMYK** - color value expressed as a value from 0 to 4,294,967,295

Colorspace

Always pass a color value appropriate to the colorspace. You can ensure this using the getPixelFormat() function in your script and then using different hexadecimal values for the RGB and the CMYK colorspaces in an IF/THEN construction. If getPixelFormat() returns "CMYK," use the CMYK value (0x plus eight more digits), and otherwise use the RGB value (0x plus six more digits).

`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:*** The `Burn` option is 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`.

> ***Important:*** If you are using smoothing for media that contains an alpha channel and you plan to save it to a format that does not support alpha channels, it is necessary to use convert() to remove the alpha channel before using this operation. Or, as an alternative, you can composite the modified image onto an opaque background before saving the image. Without this additional handling, the media will not look correct in a non-alpha file format.

`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()

The `reduce()` method applies a specified or generated color palette to the image. By default, this function generates an optimal palette of 256 colors. It accepts a boolean parameter, `UseAlpha`. It defaults to false to maintain compatibility with the old behavior. If set to true, alpha channel values below the cutoff will cause the pixels in the paletted image to be transparent.

Also an additional integer parameter, `AlphaCutoff`, can be specified. It defaults to 127. Valid values range from 0 to 255. It controls the decision as to whether to make a pixel transparent or opaque in the resulting paletted image.

If the source Media object does not have an alpha channel, the `UseAlpha` and `AlphaCutoff` parameters will be ignored.

Note: MediaRich also supports Adobe Color Table (.act) files.

Syntax

```
reduce(
[Netscape @ <true, false>]
[BW @ <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>]
[UseAlpha @ <true, false>]
[AlphaCutoff @ <0 to 255>]
[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 colorspace 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 the following:

```
MediaRichCore/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. Refer to the *MediaRich CORE Installation and Administration 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 might 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 is deprecated.

`Dither` - determines the level of dithering to use for remapping image pixels to the palette. The default is `0`, 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.

`UseAlpha` - if set to `true`, alpha channel values below the cutoff will cause the pixels in the paletted image to be transparent. It defaults to `false` to maintain compatibility with the old behavior.

`AlphaCutoff` - when `UseAlpha` is `true`, this can be specified to control the decision as to whether to make a pixel transparent or opaque in the resulting paletted image. Valid values are 0 to 255, with a default of 127.

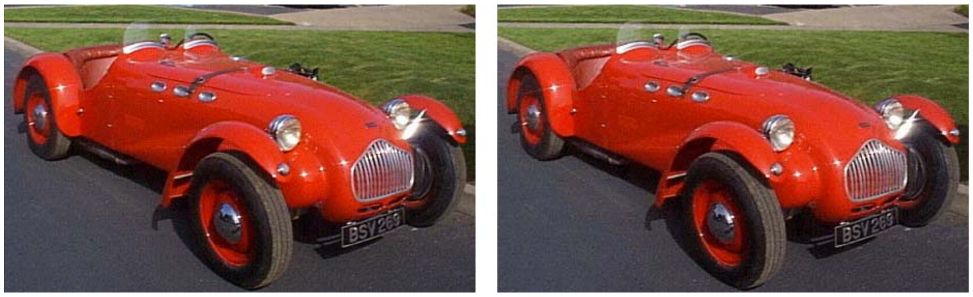
If the source Media object does not have an alpha channel, the `UseAlpha` and `AlphaCutoff` parameters will be ignored.

Note: The `FixAlpha` parameter for the `composite()` method is now depreciated and is ignored. The `fixAlpha()` method is also depreciated.

`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 137.

Example

```
var image = new Media();
image.load(name @ "car.tga");
image.reduce(colors @ 256, dither @ 3, pad @ true);
image.save(type @ "jpeg");
```



rotate()

The `rotate()` Method rotates the Media by the specified angle in degrees. This function fully supports the CMYK colorspace.

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()`. Defaults to "true" for angles of 90 and 270, and to "false" otherwise.

Note: The `Enlarge` parameter is deprecated.

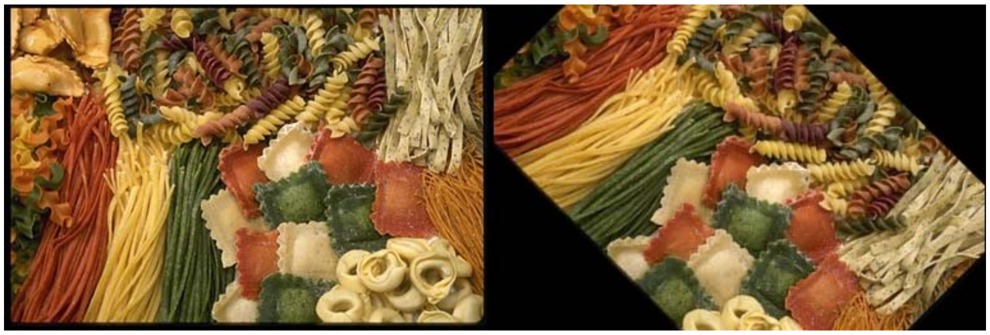
`Smooth` - provides for smooth edges when rotating to something other than right angles.

`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 137.

Example

```
var image = new Media();
image.load(name @ "pasta.tga");
image.rotate(angle @ 45, smooth @ true);
image.save(type @ "jpeg");
```



rotate3d()

The `rotate3d()` method 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 colorspace.

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 137.

Example

```
var image = new Media();
image.load(name @ "pasta.tga");
image.rotate3d(angley @ 30, distance @ 28);
image.save(type @ "jpeg");
```

save()

The `save()` method 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. The TIFF (.tif) file format supports file sizes greater than 4GB if the "Big" parameter is enabled.

Important: Loading, modifying, and saving very large image files can result in errors or crashes when the system cannot accommodate these files. If this occurs, you should first try adding more memory to your server. For more information, see "Memory Issues with Very Large Image Files" on page 362.

Also, when working with large images, ensure your page file size is set to "System Managed Size" on the enabled drive and make sure the drive has enough space to contain it.

Syntax

```
save(
[name @ "virtualfilesystem:/filename"]
[type @ <"typename">]
[embedICCProfile @ <true, false>] // (EPS, JPEG, PNG, PSD, and TIFF files only)
[SaveMetadata <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 @ <"rle", "faxg3", "faxg4", "jpeg", "lzw", "packbits", "zip",
"deflate">] // (TIFF files only)
[mode @ <"multi-resolution type”>] // (TIFF files only)
[Resolutions @ <"multi-resolution type”>] // (TIFF files only)
[TileWidth @ @ <value>] // (tiled TIFF files only)
[TileHeight @ @ <value>] // (tiled TIFF files only)
[Big @ <"On"><"Off"><"Auto">] // Big TIFF mode
[ScaleAlg @ <"Fast", "Smooth", "Outline", "Best", “OS”>] // (tiled TIFF files only)
);
```

Parameters

`name` - specifies the virtual filesystem and name for the file.

`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 colorspace: .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()` method supports ICC profiles for EPS, JPEG, PNG, PSD, and TIFF files. However, color correction for `save()` is now deprecated. The `ColorCorrect()` method is required for converting from RGB to CMYK (or visa versa). Parameters for Color Profile Specifications (`srcProfile`, `destProfile`, and `intent`) are no longer supported by `save()` method.

`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 CHAPTER 10 , “MediaRich Metadata Support” on page 326.

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 Parameter 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 Parameters for JPEG 2000 Files

`Lossless @ <true | false> (default is true)`

Turns lossless compression on or off. If it’s on (`true`), the Quality setting has no effect, since lossless compression always implies 100% quality.

`Quality @ <1..100> (default 100)`

Sets the quality level of the output image. This value is a percentage of the original uncompressed image size you wish to compress the image by. So to make the output file 3% the size of the uncompressed image, you would specify `quality @ 3`.

`Unwrapped @ <true | false> (default is false)`

Saves JPEG2000 image in the code-stream format instead of the more powerful .JP2 format.

`TargetSize @ <number in Kbytes>`

This lets you specify the size you wish the output file to be (in KBytes) and the writer will generate a file of that size, adjusting the quality as needed. If it is unable to do it, the writer will return an error. (An error is unlikely to happen except in the most extreme cases; e.g. trying to compress a 300MB file down to 1K, etc.) This option overrides the Quality setting and thus has no default value. (If it's not present, Quality @ 50 gets used.)

`Progressive @ <true|false>, default is false`

Causes a progressive image to be created using the LRCP JPEG 2000 progression. The number of progressions depends on the final quality setting: the maximum levels generated is 10. It generates levels until it reached 38% of the final quality, or 10 levels, whichever comes first. If you need more control over progressive output (more levels, specific qualities, different progression type, etc.) use the "ilyrrates" and other advanced options. See the Jasper docs for details.

> ***Note:*** It is acceptable to use TargetSize and Progressive together. You do not have to use Quality if you need to the file to be a specific size and also progressive.

`AdvancedOptions @ <"options string>"`

Specify additional options directly to the JasPer encoders. These options are added after the options above, so any options that set or change the behavior of the above options will override their settings. For more information on the settings the JasPer encoder takes, please consult the Jasper encoder documentation. (Quick ref: `ilyrrates` is used for progressive saves, which determines the compression ratio for each progression. `prg` sets the type of progression).

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 a 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 PDF Files

The PDF Writer saves the images in the Media object as rendered bitmaps embedded in a PDF file.

`JPEGCompression` (boolean, defaults to false) When `true`, the images in the PDF are saved as JPEG instead of PNG. When saving as JPEG, the JPEG writer options can also be applied. (See 'Additional Parameters for JPEG Files' for details on the available JPEG writer options.) Because of this, you can control the size of the file by adjusting the compression quality.

When saving as PNG (`JPEGCompression` is `false` or not specified), the PNG writer options can be applied (see “Additional Parameter for PNG Files” on page 180 for details).

Note: The PDF writer compresses the entire document, so the PNG compression option (when embedding a OPNG image) might not affect the file size.

When saving an RGBA-32 image (a 24-bit RGB image with an alpha channel) or an RGBA-16 image (a 15-bit RGB image with a 1-bit alpha channel) as a PDF using the default PNG compression, the image will be applied to a white background, or to the background color set in the Media object, if present. (In other words, it uses any alpha channel to apply the image to the blank page, which is white as per the PDF specification.)

If you do not require this effect, have your MediaScript remove the alpha channel data. The easiest way is to specify an `RType` of `RGB-24` as a parameter during the `save()` itself.

If you would prefer the background color to be other than white when saving RGBA-32 data to PDF with PNG compression, precede the save with a call to `setColor()` that sets '`backcolor`' to the desired color.

Important: Only RGB data with an alpha channel will be treated this way: this is not the case for CMYK+Alpha because all CMYK images are automatically switched to JPEG to maintain the CMYK colorspace, which is not supported for PNG compression, but IS supported for JPEG compression.

Additional Parameters 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` - If no compression parameter is specified, the writer picks the best, most compatible compression format. For color & grayscale images, this is `"lzw"`. For 2-color B&W images, this is `"rle"`. If the file being written is multiframe, specifying a compression parameter forces ALL frames to that compression mode. If the multiframe TIFF has both B&W and RGB images in the media, an error will occur. User should let the writer automatically select compression in such cases by not specifying the compression parameter.
- `rle` - Compresses runs of identical byte sequence values into code only a few bytes in length
- `faxg3` - Produces TIFF files that conform to the Group 3 FAX format
- `faxg4` - Produces TIFF files that conform to the Group 4 FAX format
- `jpeg` - Produces TIFF files using the JPEG compression scheme

> ***Note:*** When the JPEG 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` - Produces 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.
- `zip` - Applies a standard "zip" compression.
- `deflate` - Similar to `zip`, it applies a standard "zip" compression. However, it also stores an older tag ID in the file. This is useful for old systems that cannot use the modern "zip" compression tag.

`Resolutions` - Enables tiled TIFF mode and allows saving of subimages as a pyramid TIFF. By default, the value of this parameter is `“-1”`, which disables tiled mode. When set to `“0”`, the writer automatically calculates the number of resolutions to save as a pyramid TIFF file, until the subimage becomes smaller than the tile dimensions. When set to `“1”`, it saves a single-resolution tiled TIFF. When set to `“2”` or greater, this limits the number of resolutions to be saved or stops when the subimage becomes smaller than the tile dimensions.

`TileWidth`, `TileHeight` - These are ignored when `Resolutions` is not specified or is specified as a negative value. Use these parameters to specify the tile width and height (in pixels) for a tiled or pyramid TIFF.

`ScaleAlg` - This is ignored when `Resolutions` is not specified or is specified as a negative value. Use this parameter to specify the `Media.scale()` method to use to create the subimages for a pyramid TIFF. The default scale mode is "smooth". For more information about the valid values, see the description for the `Alg` parameter for “scale()” on page 183.

`Big` - Enables/Disables BigTIFF mode. BigTIFF is a new TIFF standard that allows for TIFF files to grow beyond 4GB in size. This parameter accepts three values:

- "`On`" - Forces BigTIFF mode to always be on; even files greater than 4GB will be saved in BigTIFF format.
- "`Off`" - (default) Forces BigTIFF mode to off. If a file grows larger than 4GB while it is being saved, an error will be reported and the save will abort.
- "`Auto`" - The file is first saved in normal TIFF format and if it cannot be saved that way, it will automatically switch to BigTIFF mode and the save will be restarted.

Save () Example

```
var image = new Media();
image.load(name @ "peppers.tif");
image.save(type @ "gif", interlaced @ true, loopcount @ 100,
removeduplicates @ true, delay @ 400);
```

saveEmbeddedProfile()

The `saveEmbeddedProfile()` method saves the profile embedded in an image to the specified disk file.

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 39 for more information.

scale()

The `scale()` method scales the image to the specified size. This function fully supports the CMYK colorspace.

Syntax

```
scale(
[Alg @ <"Fast", "Smooth", "Outline", "Best", "OS">]
[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 for scaling. This parameter supports five different algorithm modes:

- `Fast` – This is the default algorithm.
- `Smooth` - This is similar to the Fast algorithm, but produces a smoother result for scaling upward.
- `Outline` - This algorithm is designed for black and white images only.
- `Best` - The effect of this algorithm is most apparent when scaling upward -- it uses a spline algorithm, giving superior results; however, it is slower than both the Fast and Smooth algorithms.
- `OS` - This mode uses the operating system scaling to provide a fast, high-quality scale. It only works on RGB and RGBA Media rasters. Any other raster type will cause it to quietly use the `Best` mode instead. It will also fall back to `Best` if the image uses more that 512MB of memory, due to an OS limitation.

> ***Note:*** On Windows, the `OS` scaling mode sets the color of the pixels in RGBA images to black anywhere that the alpha for that pixel is completely transparent. This is different than the existing scale modes, where the color channels are left untouched. If this is not appropriate for your application of the scaled images, use the `Smooth` or `Best` scaling modes for RGBA rasters instead.

`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 pad color 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()" on page 188.

`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()` (see page 91).

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()

The `selection()` method 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 `0`, 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 colorspace.

Note: When using with two source images, both images must be the same size. This can be accomplished with the `scale()`, `getHeight()`, or `getWidth()` function. For more information, see "scale()" on page 183, "getHeight() " on page 117, and "getWidth()" on page 127.

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

`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.

- 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 wish 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 137.

Note: The `Name` parameter is now deprecated.

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()

The `setColor()` method 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()` (see page 77) and `drawText()` (see page 102), and supports the CMYK colorspace.

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, rgb, or cmyk>]
[ForeColor @ <color in hexadecimal, rgb, or cmyk>]
[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.

`Forecolor` - specifies the foreground color.

This parameter supports a hexidecimal, RGB, or CMYK color specification:

- **hexidecimal** - color value expressed as a value from 0x000000 to 0xFFFFFF (RGB colorspace) or from 0x00000000 to 0xFFFFFFFF (CMYK colorspace)
- **RGB** - color value expressed as a value from 0 to 16,777,215
- **CMYK** - color value expressed as a value from 0 to 4,294,967,295

Colorspace

Always pass a color value appropriate to the colorspace. You can ensure this using the getPixelFormat() function in your script and then using different hexadecimal values for the RGB and the CMYK colorspaces in an IF/THEN construction. If getPixelFormat() returns "CMYK," use the CMYK value (0x plus eight more digits), and otherwise use the RGB value (0x plus six more digits).

`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.

Important: You cannot specify values for both the `Backcolor` and `Backindex` parameters or for both the `Forecolor` and `Foreindex` parameters.

`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.

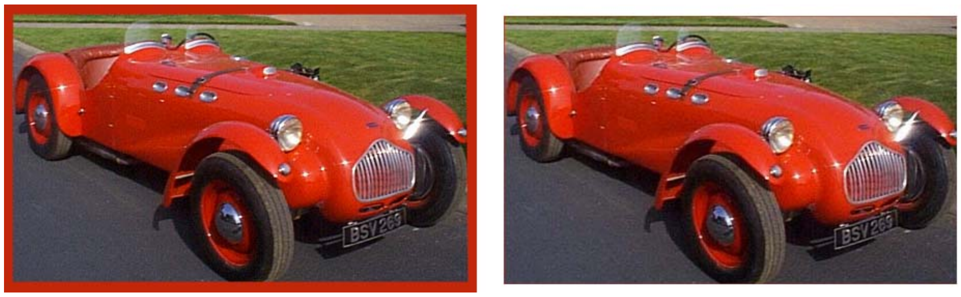
Note: The `Popular` parameter overrides any settings specified by the `Backcolor`, `Forecolor`, `Backindex` or `Foreindex` parameter. In addition, this parameter does not support the CMYK colorspace.

`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 137.

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()

The `setFrame()` method replaces the Media object for the specified frame (if available). It is common to use this function with `getFrame()` (see page 116 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()

The `setLayer()` method replaces the Media object for the specified layer (if available). In conjunction with `getLayer()` (see page 118), this is commonly used to modify layer contents before calling the `collapse()` (see page 84) 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()

The `setLayerBlend()` method 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`, `Colorize` (causes only the hue component of the source to be stamped down on the image).

Note: The `Burn` option is deprecated. `ColorBurn` results in the same blend.

Example

```
var image = new Media();
image.setLayerBlend(2, "Difference");
```

setLayerEnabled()

The `setLayerEnabled()` method 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 (equivalent to 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()

The `setLayerHandleX()` method 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 attachment point on the x-axis for the selected layer. The default is `Center`. Other options are `Left` and `Right`.

Example

```
var image = new Media();
image.setLayerHandleX(2, "Right");
```

setLayerHandleY()

The `setLayerHandleY()` Method 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()

The `setLayerOpacity()` method 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()

The `setLayerPixels()` method 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(
<layerIndex>,
<media>
);
```

Parameters

`layerIndex` - specifies the layer in the target image that gets its pixels replaced. The default is the first layer (starting from zero).

`media` - specifies the source Media object.

> ***Important:*** 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()

The `setLayerX()` method 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 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()

The `setLayerY()` method 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 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);
```

setMetadata()

The `setMetadata()` method 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()

The `setPixel()` method sets the color of one pixel to the chosen color value. This works in both RGB and CMYK colorspaces.

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 188.

`layers` - for PSD files, specifies the layers to be affected. The layer numbers begin at 0 (background). For more information see "load()" on page 137

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 colorspace), 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()

The `setResolution()` method changes the DPI of the image in memory and fully supports Media objects within the CMYK colorspace.

Note: This information can be stored only in the following formats: BMP, EPS, 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 axes only.

Note: When the `Dpi` parameter and one or both of the single axes 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 137.

Example

```
var image = new Media();
image.load(name @ "peppers.tga");
image.setResolution(dpi @ 300, xdpi @ 200);
image.save(type @ "jpeg");
```

setSourceProfile()

The `setSourceProfile()` method 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 86. For more information about color management, see “MediaRich Color Management” on page 318.

sharpenSharpen()

The `sharpenSharpen()` method makes the edges in the image more pronounced.

> ***Note:*** This function is “selection aware”—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see “selection()” on page 186.

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()

The `sharpenSharpenMore()` method sharpens the clarity of an image. This is similar to the `sharpenSharpen()` function, but to a greater extent.

Note: This function is “selection aware”—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see “selection()” on page 186.

Syntax

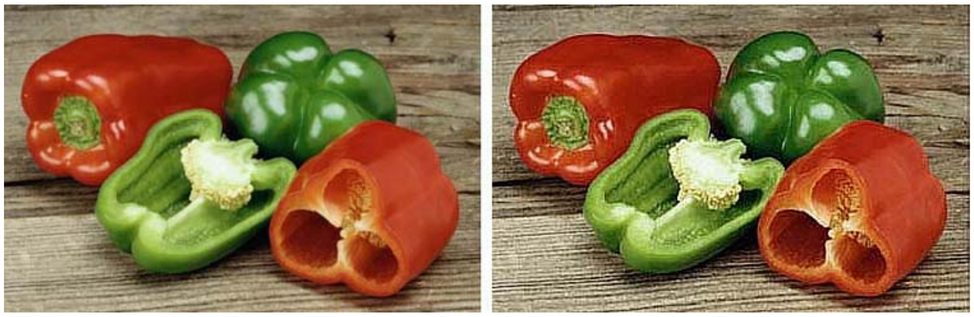
```
sharpenSharpenMore();
```

Parameters

This function has no parameters.

Example

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



sharpenUnsharpMask()

The `sharpenUnsharpMask()` method 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"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

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()

The `sizeText()` method returns the width, height, and lines for the specified parameters.

Note: 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 *local.properties* file to change the default fonts directory. Refer to the *MediaRich Installation and Administration 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 or Linux.

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

- “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 an .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()

The `stylizeDiffuse()` method applies a filter that makes the image appear as though viewed through a soft diffuser, with options to lighten or darken the effect.

Syntax

```
stylizeDiffuse(
[Radius @ <value 0..10000>]
[Mode @ <"mode"]
);
```

Note: This function is "selection aware"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

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()

The `stylizeEmboss()` method applies a filter that makes the image appear as though embossed on paper.

> ***Note:*** This function is "selection aware"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

Syntax

```
stylizeEmboss(
[Height @ <value 1..10>]
[Angle @ <value -360..360>]
[Amount @ <value 1..500>]
);
```

Parameters

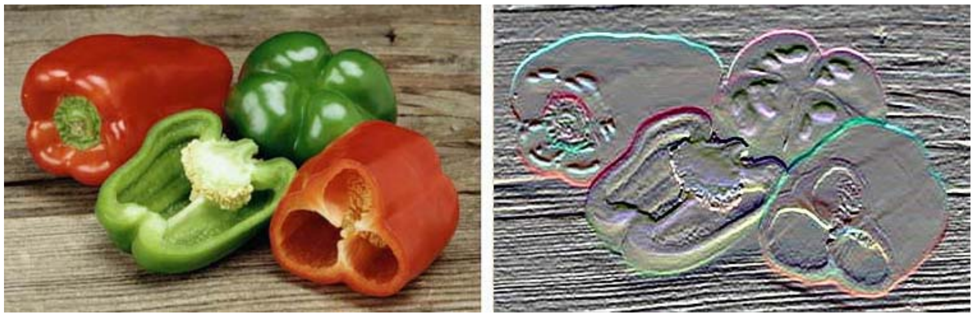
`Height` - determines the depth of the embossing effect. The default is `3`.

`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()

The `stylizeFindEdges()` method traces the edges (areas of significant transitions) of the image with broad lines.

Note: This function is “selection aware”—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see “selection()” on page 186.

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`.

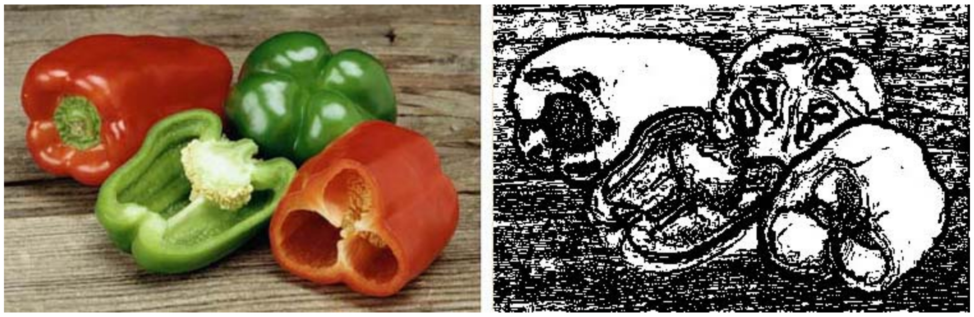
`Grayscale` - produces a monochromatic result. The default is `false`.

`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()

The `stylizeTraceContour()` method 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"—if a selection is made, the system applies the function based on the current selection. For more information about making selections, see "selection()" on page 186.

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()

The `zoom()` method zooms in on a specified portion of the Media object and fits it to the specified size. This is equivalent to a crop followed by a scale. This function fully supports the CMYK colorspace.

Syntax

```
zoom(
[Alg @ <"Fast", "Smooth", "Outline", "Best">]
[Fit @ <"Full", "Width", "Height">]
[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.

`Fit` - specifies the sizing type for the Media. The default is `Full`.

`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 when `Fit` is set to`Full`. Otherwise, it is fitted to `Xs` when `Fit` is set to `Width`, or fitted to `Ys` when `Fit` is set to `Height`.

`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`. specify

`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 137.

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");
```

File Object

The File object provides access to a file or directory. The File object is useful if you need information about a file. For example, you would use the File object when you need to check the file before loading the contents of the file.

File() constructor

The File object is constructed using the `File()` constructor.

Syntax

```
var fileObject = new File(<filePath> );
```

Parameters

`filename` - a string containing the name (and optionally the path) of the file with which the object is to be associated. If the string does not specify a file system, the default is the write file system. See "File Systems" on page 39 for more information.

File object methods

- clear()
- clearCached()
- close()
- copy()
- exists()
- freeSpaceGB()
- getFileExtension()
- getFileName()
- getFileNameNoExt()
- getFilePath()
- getFilePathNoExt()
- getLastAccessed()
- getLastModified()
- getParentPath()
- getSize()
- getType()
- isDirectory()
- isFile()
- isShared()
- length()
- list()
- mkdir()
- read()
- readAll()
- readBinary()
- readNextLine()
- remove()
- rename()
- rmdir()
- setShared()
- write()
- writeBinary()

clear()

The `clear()` method clears the contents of the file pointed to by the file object, if the file exists.

Syntax

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

Parameters

This function takes no parameters.

clearCached()

The `clearCached()` method causes any cached version of a file to be discarded. Provides a solution for temporarily bad images that make it into the FSNet filesystem's cache.

Syntax

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

Example

```
var filepath = "http://www.google.com/intl/en/images/logo.gif";
try
{
// Try to open the image
image.load(name @ filepath);
}
catch (ex)
{
// The open failed, so clear the cache and try again
var f = new File(filepath);
f.clearCached();
image.load(name @ filepath);
}
```

Parameters

This function takes no parameters.

close()

The `close()` method closes the file and flushes any output immediately.

Syntax

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

Parameters

This function takes no parameters.

copy()

The `copy()` method copies the contents of the file to a new file named “destFile”. The object’s original filename is unchanged.

Syntax

```
<object name>.copy(<"destFile"> [, startPosition] [, length]);
```

Parameters

`destFile` - a string containing the name of the destination file. If the string does not specify a file system, the default is the write file system. See “File Systems” on page 39 for more information.

`startPosition` - an optional 64-bit value that sets the copy to begin reading the source file from that byte offset. If not specified, the starting position is the start of the file.

`length` - an optional 64-bit value that indicates how many bytes to copy to the destination file. If not specified, the default behavior is to copy the remaining bytes in the source file.

exists()

The `exists()` method returns `true` if the File object points to an existing file or directory; otherwise, it returns `false`.

Syntax

```
var fileExists = <object name>.exists();
```

Parameters

This function takes no parameters.

freeSpaceGB()

The `freeSpaceGB()` method returns a floating point number specifying free space in gigabytes on the volume containing the file.

Syntax

```
var freeSpaceinGB = <object name>.freeSpaceGB();
```

Parameters

This function takes no parameters.

getFileExtension()

The `getFileExtension()` method returns the extension of the file name portion of the object.

Syntax

```
var ext = <object name>.getFileExtension();
```

If the object file path is:

`Equilibrium/MediaRichCore/Shared/Originals/Media/camera.png`

`getFileExtension()` returns: `.png`

Parameters

This function takes no parameters.

getFileName()

The `getFileName()` method returns the filename portion of the object's path.

Syntax

```
var fileName = <object name>.getFileName();
```

If the object's file path is:

`Equilibrium/MediaRichCore/Shared/Originals/Media/camera.png`

`getFileName()` returns: `camera.png`

Parameters

This function takes no parameters.

getFileNameNoExt()

The `getFileNameNoExt()` method returns the filename portion of the object's path, without extension.

Syntax

```
var fileNameNoExt = <object name>.getFileNameNoExt();
```

If the object file path is:

`Equilibrium/MediaRichCore/Shared/Originals/Media/camera.png`

this function returns: `camera`

Parameters

This function takes no parameters.

getFilePath()

The `getFilePath()` method returns the full file path for the object.

Syntax

```
var filePath = <object name>.getFilePath();
```

If the object is created as:

```
var f = new File("camera.png");
```

this method returns: `write:/camera.png`

Parameters

This function takes no parameters.

getFilePathNoExt()

The `getFilePathNoExt()` method returns the file path for the object with the extension removed.

Syntax

```
var filePathNoExt = <object name>.getFilePathNoExt();
```

If the object is created as:

```
var f = new File("camera.png");
```

the method returns: `write:/camera`

Parameters

This function takes no parameters.

getLastAccessed()

The `getLastAccessed()` method 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

```
var accTimeInSecs = <object name>.GetLastAccessed();
```

Parameters

This function takes no parameters.

getLastModified()

The `getLastModified()` method returns the last modified date as seconds since midnight January 1, 1970. If the file does not exist or cannot be accessed, the function returns `0`.

Syntax

```
var modTimeInSecs = <object name>.getLastModified();
```

Parameters

This function takes no parameters.

To use the return value as an argument to new Date(), multiply it by 1000:

```
var modDate = new Date(modTimeInSecs * 1000);
```

getParentPath()

The `getParentPath()` method returns the parent path for the object.

Syntax

```
var parentPath = <object name>.getParentPath();
```

If the object file path is the following:

```
write:/camera.png
```

the method returns: write:/

Parameters

This function takes no parameters.

getSize()

The `getSize()` method returns the file size in bytes. If file does not exist, cannot be accessed, or is empty, the function returns `0`.

Syntax

```
var sizeInBytes = <object name>.getSize();
```

Parameters

This function takes no parameters.

getType()

The `getType()` method returns the type name of this object, which is “File”.

Syntax

```
var objectTypeName = <object name>.getType();
```

Parameters

This function takes no parameters.

isDirectory()

The `isDirectory()` method returns `true` if the File object points to a directory; otherwise, it returns `false`.

Syntax

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

Parameters

This function takes no parameters.

isFile()

The `isFile()` method returns `true` if the File object points to a regular file; otherwise, it returns `false`.

Syntax

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

Parameters

This function takes no parameters.

isShared()

The `isShared()` method returns `true` if the File object is shared. This would be the case if setShared() (see page 219) was called for the object to allow shared write.

Syntax

```
var isShared = <object name>.isShared();
```

Parameters

This function takes no parameters.

length()

The `length()` method returns an integer containing the number of text lines in the file.

Syntax

```
var numLines = <object name>.length();
```

Parameters

This function takes no parameters.

list()

If the File object represents a directory, the `list()` method returns an array of File objects representing the directory entries. If the named File object does not represent a directory (or the directory is empty), it returns an empty array.

Syntax

```
var files = <object name>.list();


for (var f in files)
print(files[f].getFileName() + "\n");
```

Parameters

This function takes no parameters.

mkdir()

The `mkdir()` method 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()

The `read()` method reads the specified line number from the file and returns it as a string. If the specified line is not found, the method returns `undefined`.

> ***Note:*** Line numbers start at zero (0), not at one (1).

Syntax

```
var line = <object name>.read( <index> );
```

Parameters

`index` - specifies the line number and can range from 0 to number-of-lines - 1.

readAll()

The `readAll()` method reads the entire contents of the file and returns it as a string.

Syntax

```
var fileContents = <object name>.readAll( );
```

Parameters

This function takes no parameters.

readBinary()

The `readBinary()` method reads the specified number of bytes from the file and returns it as a buffer/string object. When the end of the file is reached, the function will continually return an array of 0 bytes.

The starting position can be specified in situations where the script does not need to read from the start of the file. The returned buffer can be treated as binary data using the specialized Buffer object methods or simply used as a string.

Syntax

```
var buffer = <object name>.readBinary(<length> [, position]);
```

Parameters

`length` - specifies the number of bytes to read from the file. If you read past the end of the file, the request will truncate the data to the end of the file.

`position` - specifies the byte offset into the file to read the data from. If you specify a position past the end of the file, an empty buffer object is returned. If not specified, the file is read from the current file position.

Example

```
// Read the TIFF architecture identifier from an image and convert it into a string.
// This example is more complex than it has to be in order to demonstrate a couple of
the Buffer object methods.
var f = new File("img.tif");
var buf = f.readBinary(16, 0); // read the TIFF header from the start of the file
if (buf.size >= 2)
var arch = buf.subBuffer(0, 2).toString(); // extract the arch id as a string (will
be "II" or "MM")
```

readNextLine()

The `readNextLine()` method reads the next line from the file referenced by the File object, and returns a string containing the text in that line. It returns `undefined` at the end of the file.

Syntax

```
var line = <object name>.readNextLine();
```

Parameters

This function takes no parameters.

remove()

The `remove()` method deletes the file referenced by the File object.

Syntax

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

Parameters

This function takes no parameters.

rename()

The `rename()` method renames the file referenced by the File object (but it does not rename the object).

Syntax

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

Parameters

`newname` - a string containing the new file name.

rmdir()

The `rmdir()` method 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` - Boolean - if `true`, the directory and all of its content is removed. If `false`, the directory is only deleted if it is empty.

setShared()

The `setShared()` method declares a file sharable, so multiple threads can write to it.

Syntax

```
<object name>.setShared( <shared> );
```

Parameters

`shared` - Boolean - if `true`, the file will be sharable from now on. If `false`, the file will no longer be sharable.

write()

The `write()` method writes a string to the end of the file.

Syntax

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

Parameters

`string` - string to write.

writeBinary()

The `writeBinary()` method writes a binary block to the file.

Syntax

```
<object name>.writeBinary( <blob>);
```

Parameters

`blob` - an object containing binary data.

The most common use is to write binary objects created by QuickTime.

System Object and Methods

The System object provides access to system information. For example, you might need to retrieve information about the system to determine support for another object type function.

This object implements the following methods:

Method	Usage
`fontExists()`	Returns `true` if a font with the specified name exists It takes the font name as its only parameter
`getCPUName()`	Returns the name of the CPU architecture
`getFontList()`	Returns an array containing the names of all fonts known by the system
`getFreeMemory()`	Returns the amount of free system memory ***Example:*** `print("System free memory is " + System.getFreeMemory() + " bytes.\n");` Note: Available only in Windows.
`getOSName()`	Returns the name of the operating system
`getTotalMemory()`	Returns the amount of total system memory ***Example:*** `print("System total memory is " + System.getTotalMemory() + " bytes.\n");` Note: Available only in Windows.
`sendScriptStatus (<evt>)`	Publishes the specified evt string to all subscribers Note: There must be some way to subscribe to these events, and there must be some list of what events are in use.
`resetScriptTimer`	Resets the script timer
`pushScriptTimeout (<time>)`	Changes the script timeout to time, saving the current value on the stack
`popScriptTimeout()`	Restores the script timeout to the value from the stack If the stack is empty, the timeout is unchanged. Note: `resetScriptTimer()` should be called before this method if the tos is smaller than the current value. Otherwise the script will timeout immediately.

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, refer to http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/

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 "The MediaScript #include Directive" on page 37.

The `XmlDocument` object is constructed using the `new XmlDocument()` constructor.

Syntax

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

Methods

The `XmlDocument` object has all the methods of the DOM's Document class, as well as the following methods:

- loadFile()
- loadString()
- save()

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;
```

loadFile()

The `loadFile()` method 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 its only parameter. By default, MediaRich looks for XMLDocument files in the write file system which points to the following directory:

```
MediaRichCore/Shared/Originals/Media
```

> ***Note:*** You can modify the MediaRich server *local.properties* file to change the default Media directory. See "File Systems" on page 39 for more information.

MediaRich allows supports setting virtual file systems and can load files from that location. Virtual file systems are defined in the MediaRich server *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()

The `loadString()` method 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()

The `save()` method 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 that points to the following directory:

```
MediaRichCore/Shared/Originals/Media
```

> ***Note:*** You can modify the MediaRich server *local.properties* file to change the default Media directory. See “File Systems” on page 39 for more information.

MediaRich also supports setting virtual file systems and then save files to that location. Virtual file systems are also defined in the MediaRich server *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");
```

TextResponse Object

The TextResponse objects allow users to create response objects that take strings and save text files (plain text, html, or xml). There are no properties for this object.

The TextResponse object is 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);
}
```

Methods

The TextResponse object implements the following methods:

- append()
- getText()
- getTextType()
- getType()
- setText()
- setTextType()

append()

The `append()` method 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()

The `getText()` method returns the text associated with the named object.

Syntax

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

Parameters

This function has no parameters.

getTextType()

The `getTextType()` method returns the text type associated with this object: `TextResponse.TypePlain`, `TextResponse.TypeHtml`, or `TextResponse.TypeXml`.

For more information about text types, see "TextResponse Object" on page 223.

Syntax

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

Parameters

This function has no parameters.

getType()

The `getType()` method returns the type of the named TextResponse object, which is always `TextResponse`.

Syntax

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

Parameters

This function has no parameters.

setText()

The `setText()` method 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()

The `setTextType()` method sets the text type associated with this object.

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

TextExtraction Object

The TextExtraction object allows users to extract text from the file types indicated in the Image and Office document tables in the *MediaRich CORE Installation and Administration Guide* ("File Format Support" appendix).

TextExtraction object property

`VisualIntegrity.UseUnicode`

This defaults to true if not present in any of the local properties files. When true, the default behavior of `TextExtraction.getEntireText()` is to return Unicode character strings. If false, the text is forced to ASCII.

Note: This setting only affects PDF and AI text extraction—Office text extraction is always Unicode and EPS/PS text extraction is always ASCII. However, ASCII text extraction is possible for presentation-type formats, such as PowerPoint.

The TextExtraction object is constructed using the `TextExtraction()` constructor.

Syntax

```
var strObject = TextExtraction(<filePath>);
```

Parameters

`filePath` - the path to the file from which text is to be extracted.

Methods

The TextExtraction object implements the following methods:

- getEntireText([useUnicode])
- getText()

getEntireText([useUnicode])

The `getEntireText` method provides Unicode text extraction for PDF/EPS/PS/AI/Office documents. If not specified, it defaults to whatever is defined in the `VisualIntegrity.UseUnicode` property in the local/user properties file. If it is specified, it overrides the property value.

Note: To specify it on a per-file basis, the `Media.load()` accepts a `useUnicode` parameter for those document types.

If true, it returns the text associated with the named object. The returned text is one long continuous ASCII string of text, with linefeeds and all other characters.

If false, it returns ASCII.

The default setting can be specified in the properties file:

```
VisualIntegrity.UseUnicode=true
```

To set this property, add this line to the local properties (or user properties) file and then restart MediaRich.

> ***Note:*** When Unicode text extraction is used, paged text extraction on Windows is not available and only `TextExtraction.getEntireText()` will work. (Other class methods will return an error.) Mac and Linux do not support paged text extract, so the issue is not relevant on those platforms.

Syntax

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

Parameters

`UseUnicode` - An optional boolean parameter. If true, it returns Unicode text. If false, the text is forced to ASCII. If this parameter is not present, the default is whatever the VisualIntegrity.UseUnicode property is set to (which defaults to true). This parameter only affects the text of PDF and AI files. Office files always return Unicode text, and EPS/PS files are always ASCII.

Example

```
var te = new TextExtraction("text-extraction.pdf");
var str = te.getEntireText();
var m = new Media();
m.makeCanvas(xs @ 1024, ys @ 768, fillcolor @ 0xffffff);
m.makeText(text @ str, font @ "Arial Unicode MS", style @ "Bold",
size @ 14, color @ 0x000000);
m.save(name @ "out.png");
```

getText()

The `getText()` method returns the text associated with the named object within a given page range. It returns the text as one long continuous string of text, with linefeeds and all other characters.

> ***Important:*** This functions does not currently work with Office documents and is also disabled for PDF/EPS/AI files when Unicode mode is turned on. (Unicode paged text extraction is currently unsupported.)

Syntax

```
<object name>.getText(page1, pageN);
```

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

Parameters

`page1` is the first page, and (optional) `pageN` is the last page, from which getText extracts text.

Example

```
function main() {
var te = new TextExtraction(filePath);
var page2text = te.getText(2);
var page2to5text = te.getText(2, 5);
}
```

IccProfile Object

The IccProfile object is constructed using the `IccProfile()` constructor. It 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 example 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);
```

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 39 for more information.

The list(colorspace, class) method is a static method of the IccProfile class.

The IccProfile object implements the following methods:

- close()
- getName()
- getPath()
- getClass()

- getColorspace()
- getConnectionspace()

IccProfile.dll

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.

The following example 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);
}
```

list(colorspace, class)

The list(colorspace, class) method is a 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 example returns an array containing all the ICC monitor and printer profiles for RGB and CMYK colorspaces:

```
#link "IccProfile.mdv"
var profs = IccProfile.list(IccProfile.RGB | IccProfile.CMYK, IccProfile.MONITOR |
IccProfile.PRINTER);
```

close()

The `close()` method closes the profile file. It returns no value.

Parameters

This function takes no parameters.

Example

The following example 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();
```

getName()

The `getName()` method returns the friendly display name of the profile.

Parameters

This function takes no parameters.

Example

The following example creates an IccProfile object using the USWebCoatedSWOP.icc profile and retrieves the friendly name:

```
#link "IccProfile.dll"
var prof = new IccProfile("USWebCoatedSWOP.icc");
var name = prof.getName();
```

getPath()

The `getPath()` method returns the path of the profile file.

Parameters

This method has no parameters.

Example

The following example creates an IccProfile object using the USWebCoatedSWOP.icc profile and retrieves the profile path:

```
#link "IccProfile.dll"
var prof = new IccProfile("USWebCoatedSWOP.icc");
var name = prof.getPath();
```

getClass()

The `getClass()` method returns the profile class.

Parameters

This method has no parameters.

Example

The following example 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()

The `getColorspace()` method returns the profile colorspace.

Parameters

This function takes no parameters.

Example

The following example 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()

The `getConnectionspace()` method returns the IccProfile connection space as the `jseNumber`.

Parameters

This function takes no parameters.

Example

The following example creates an IccProfile object using the USWebCoatedSWOP.icc profile and retrieves the `connectionspace`.

```
#link "IccProfile.dll"
var prof = new IccProfile("USWebCoatedSWOP.icc");
var connectionspace = prof.getConnectionspace();
```

Zip Object

The Zip object is used to create and add files to a new zip archive. The Zip object is constructed using the `Zip()` constructor. This function takes no parameters.

> ***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 39 for more information.

This object implements the following methods:

- addFile()
- save()

addFile()

The `addFile()` method adds a file to the zip archive.

> ***Note:*** The file is not actually read until the `save()` method is called. For more information, see "save()" on page 233.

Parameters

`filePath` - specifies the VFS path of file to add to archive.

`archivePath` - specifies a full path of file as stored in archive.

save()

The `save()` method creates a new zip archive and compresses all the files specified by calls to `addFile()`. For more information, see "addFile()" on page 233.

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 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 and is constructed using the `Unzip()` constructor. This function takes no parameters.

> ***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 39 for more information.

This object implements the following methods:

- close()
- extractAll()

close()

The `close()` method closes the archive file.

Parameters

This function takes no parameters.

Examples

```
var myZip = new Unzip();
myZip.open("zip/files.zip");
myZip.extractAll("files");
myZip.close();
```

The following example extracts 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();
```

extractAll()

The `extractAll()` method extracts all files in the zip archive to the specified directory. The full paths stored in the archive are preserved in the destination directory.

Parameters

`dir` - specifies the VFS path of the directory to extract files.

extractFile()

The `extractFile()` method extracts the current file to the specified file.

Parameters

`destFile` - specifies the VFS path of the destination file.

firstFile() Method

The `firstFile()` method resets file iterator to first file in archive. It returns a string with the name of current file, or null if not found.

Parameters

This function takes no parameters.

getFileName()

The `getFileName()` method returns name of current file in the archive.

Parameters

This function takes no parameters.

nextFile()

The `nextFile()` method advances file iterator to next file in the archive. It returns a string with the name of the current file, or null if not found.

Parameters

This function takes no parameters.

open()

The `open()` method opens an existing archive. The archive remains open until one of the following occurs:

- Another archive is opened
- The archive is explicitly closed with the `close()` method
- The Unzip object is garbage collected

Parameters

`archiveName` - specifies the path to the existing archive file.

Emailer Object

The Emailer object is used to send email. After the message is constructed usig object methods, the object logs in to specified SMTP server using specified credentials and sends the message. The message is send as multi-part message and can contain plain text and HTML parts. Both parts are encoded as 8-bit/UTF-8, so it is Unicode-safe.

To use this object, you have to load a library that implements it with this statement at the top of your script:

```
#link <EMailer>
```

The Emailer object is constructed using the `Emailer()` constructor. This function takes no parameters.

This object implements the following methods:

- addAttachment()
- addBccAddress()
- addCcAddress()
- addToAddress()
- send()
- setFromAddress()
- setMessage()
- setMessageHTML()
- setPassword()
- setServer()
- setSubject()
- setUsername()

addAttachment()

The `addAttachment()` method adds a file attachment to the email message.

Parameters

`filePath` - path to the file to be attached.

addBccAddress()

The `addBccAddress()` method adds an email address to the list of “BCC:” addressees.

Parameters

`bccAddress` - specifies the “BCC:” receipient email address(es).

addCcAddress()

The `addCcAddress()` method adds an email address to the list of “CC:” addressees.

Parameters

`ccAddress` - specifies the “CC:” receipient email address(es).

addToAddress()

The `addToAddress()` method adds an email address to the list of “To:” addressees.

Parameters

`toAddress` - specifies the “To:” recipient email address(es).

send()

The `send()` method sends the message.

Parameters

This function takes no parameters.

Example

```
#link <EMailer>

var msg = new EMailer();
msg.setServer("smtp.coolwidgets.com:26");
msg.setUsername("joe");
msg.setPassword("mysecretpassword");
msg.setFromAddress("joe@coolwidgets.com");
msg.addToAddress("jim@hotwidgets.com");
msg.setSubject("Our new widget is cool");
var link = “http://coolwidgets.com/newwidget.html”;
var bodyText = “Check out our new widget. It’s cool:\n”;
bodyText += link + “\n\n”;
msg.setMessage(bodyText);
var bodyHTML = “<html><head><title>Cool New Widget</title></head>”;
bodyHTML += “<body><table><tr><td>\n”;
bodyHTML += “<a href=\"" + link + "\">Our new widget</a>\n";
bodyHTML += "</td></tr></table></body></html>\n";
msg.setMessageHTML(bodyHTML);
msg.send();
```

setFromAddress()

The `setFromAddress()` method sets the “From:” header in the message.

Parameters

`fromAddress` - specifies the sender email address (such as username@hostname.net).

setMessage()

Sets plain text part of the message.

Parameters

`plainText` - specifies the plain text part of the email body.

Example

```
#link <EMailer>

var msg = new EMailer();
...
msg.setMessage(bodyText);
var bodyHTML = “<html><head><title>Important Message</title></head>”;
bodyHTML += “<body><table><tr><td>\n”;
bodyHTML += “<a href=\"" + link + "\">Policy Update</a>\n";
bodyHTML += "</td></tr></table></body></html>\n";
msg.setMessageHTML(bodyHTML);
....
```

setMessageHTML()

Set HTML part of the message.

Parameters

`htmlText` - HTML part of the email body.

Example

```
#link <EMailer>

var msg = new EMailer();
...
msg.setMessage(bodyText);
var bodyHTML = “<html><head><title>Cool New Widget</title></head>”;
bodyHTML += “<body><table><tr><td>\n”;
bodyHTML += “<a href=\"" + link + "\">Our new widget</a>\n";
bodyHTML += "</td></tr></table></body></html>\n";
msg.setMessageHTML(bodyHTML);
....
```

setPassword()

If the server requires authentication, log in using specified password.

Parameters

`password` - specifies the login password for SMTP server.

Example

```
#link <EMailer>

var msg = new EMailer();
...
msg.setPassword("mysecretpassword");
....
```

setServer()

Tells the object to use specified server for sending email.

Parameters

`serverNameOrIP` - DNS name or IP address of the SMTP server.

Example

```
#link <EMailer>

var msg = new EMailer();
...
msg.setServer("smtp.coolwidgets.com:26");
....
```

setSubject()

Sets the “Subject:” line of the email message.

Parameters

`subject` - specifies the email subject text.

Example

```
#link <EMailer>

var msg = new EMailer();
...
msg.setSubject("Read this message");
....
```

setUsername()

If the server requires authentication, log in using specified username.

Parameters

`username` - login user name for SMTP server.

Example

```
#link <EMailer>

var msg = new EMailer();
...
msg.setUsername("JaneDoe");
....
```

Cubic Object (2D Interpolator)

Given two arrays, one containing X values and the other containing Y values, the Cubic object constructs a cubic spline between the supplied points and then lets you map X values to Y values anywhere on the spline.

This object supports the following constructor and method.

Constructor

The `Cubic(numPoints, Xvalues, Yvalues)` constructor takes two arrays: X values and Y values and constructs cubic spline between the supplied points.

Method

The `getValue(x)` method maps X value to Y value on the spline.

Example

```
var xValues2 = [ 0, .2, .5, .8, 1 ];
var yValues2 = [ 0, .1, .5, .9, 1 ];  // slow start/end
var cubic2 = new Cubic(5, xValues2, yValues2);

// animate a 20 step move in a full circle, with an ease in/out
// using that curve
for (i = 0 ; i <= 1 ; i += .05)
angle = 360 * cubic2.getValue(i);
```

The RgbColor Object

The RgbColor object is constructed from a 24-bit value. It splits the color into red, green, and blue components.

This object supports the following constructor, properties, methods, and static functions.

Constructor

The `RgbColor(value)` constructor 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

The `valueOf()` method converts the red, green, and blue components back to a 24-bit value.

The `toString()` method returns string representation of 24-bit value.

Static functions

The `RgbColorFromRGB(red, green, blue)` function constructs RgbColor from the components.

The `RgbColorFromPct(red, green, blue)` function 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);
```

The CmykColor Object

The CmykColor object is constructed from a 32-bit value. It splits the color into cyan, magenta, yellow, and black components.

This object supports the following constructor, properties, methods, and static functions.

Constructor

The `CmykColor(value)` constructor returns a 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.

Static 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);
```

The AWS Object

Instead of adding the new methods to an existing object, an entirely new host object was created, the AWS Object.

In order for the generated URLs to be valid in AWS, the user must place a plain text file named "credentials" (no quotes and no file extension) in a folder named .aws which should be created in the user hive folder of the user MediaRich is running as. (Create the folder with mkdir in a command line; the Windows Explorer environment won't allow the creation of a folder with a leading period in the name.) If it is LocalSystem, it will be in the Windows/System32/config/systemprofile directory. If MediaRich is running as a different user, their hive folder is commonly found under C:\Users\<mediarichuseraccountname>. A credentials files looks like this…

```
[default]

aws_access_key_id =

aws_secret_access_key =
```

…where the values for the access and secret keys follow each equal sign and are determined when an IAM user on AWS is created. Alternatively, two environment variables, `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, can also have the values assigned to them for use in MediaRich. Refer to the AWS documentation for all details of access keys and secret keys.

Adding environment variables on Windows 10:

Control panel - System and Security - System - Advanced system settings - Environment Variables... - System variables - New...

To check if your MediaRich runs as Local System:

Control panel - System and Security - Administrative Tools - Services

Note: In Task Manager - Details, it will show as "SYSTEM"

These credentials must be configured before you generate the URLs. If the URLs are generated before the access and secret is defined, the URLs will not be valid when used with MediaRich I/O operations on AWS (available through `Media.load()` and `Media.save()`).

Once this is done, the user must add `#link "AWS.mdv"` to their script to make use of the following new methods:

```
AWS.createS3GetURL(RemotePath, TimeToLive, Bucket, Region)
```

Create a limited-time S3 URL for use with Media.load() or other read operations in MediaRich. This method does no actual I/O - it only generates a URL. If the information input here is not correct, the read operation will fail. Note that you must specify your AWS IAM user's access key and secret elsewhere (it is insecure to have these values as plain text in your MediaScript). Please see the docs for the locations of where these keys will be searched.

Params

`string RemotePath` - The path and/or filename for the remote file that is desired to be saved and uploaded to the S3 storage. Do not include the bucket in the path - it is a separate parameter. If subdirectories are included and don't exist on S3 they will be automatically created.

`int TimeToLive` - The number of seconds the URL will be considered valid by the remote S3 server, measured in seconds. This time-to-live starts counting down from the moment the URL is created by this method, so leave enough time for any other script processing occurring after this call as well as the write operation itself to complete when choosing a value. As of this writing, AWS limits this value to a maximum of 7 days, so specifying values larger than 604,800 seconds will be of no use.

`string Bucket` - Specify the bucket to access. This parameter is optional; if it is left off the bucket will be read from local.properties. If it isn't present there and isn't included as a param, an error will occur.

`string Region` - Indicate the geographical region of the S3 server this URL with be used with. This parameter is optional; if it is left off the region info will be read from local.properties. If it isn't present there in this situation, an error will occur.

The names of the keys for the global bucket and region values in local.properties are AWS.S3.Bucket and AWS.Region, respectively. (They are case-sensitive as all MediaRich properties are.)

Returns a string containing the generated URL. If an error occurs while trying to generate it the URL may be invalid or empty.

```
-------------EXAMPLE HOT FOLDER SCRIPT: START-------------
// HF_ERROR_MAILLIST =
// HF_RESULTS_MAILLIST =
// HF_DELETE_ORIGINALS = false
/*
INTENT: Read from and write to AWS. The read portion is just one image that will be
composited over all images dropped.
NOTES:
BatchID ^Processed subfolders will still be created locally, but not in the ^Results
folder. The Batch ID folder will be created in the AWS location, and will contain the
processed images.
Any .errors files will appear in local ^Results subfolders; they do not appear at the
AWS location.
As written, this MediaScript does not reproduce the original folder if a folder of
images is dropped: it writes only the individual images to the AWS location inside
the BatchID folder.
```

```
If the image filename and the folder's BatchID name are duplicated by this script,
the original image will be overwritten.
*/
// REQUIRED for AWS S3 operations
#link "AWS.mdv"
function hf_file(context)
{
// Create a secure URL to a brush image on AWS.
var gUrl = AWS.createS3GetURL("32bitcomposnblur.png", 130, "test.equilibrium.com",
"us-west-1");
// Load the source image using loadAsRgb() to match the RGB
// data in the brush image and thus avoid errors.
var image = new Media();
image.loadAsRgb(name @ context.getSourcePath());

// Load the brush image.
var brush = new Media();
brush.load(name @ gUrl);
// Composite brush over each incoming image.
image.composite(source @ brush);
// Get the name (minus extension) of each image.
var sourcePath = context.getSourcePath();
var f = new File(sourcePath);
var fn = f.getFileNameNoExt();
// Stitch together the path, name & extension to be saved.
var outPath = "output" + "/" + context.getBatchId() + "/" + fn + ".jpg";
// Create a secure URL for each image to be saved to AWS.
var pUrl = AWS.createS3PutURL(outPath, 30, "test.equilibrium.com", "us-west-1");
// Save the result to the PerBatch output directory, with the
// result named the same as the source file, saved as a JPEG.
image.save(name @ pUrl);
}
function hf_post(context)
{
context.setDeleteOriginals(context.getParameter("HF_DELETE_ORIGINALS") == "true");
}
--------------EXAMPLE HOT FOLDER SCRIPT: END--------------
```

The Azure Object

MediaScript:

In order to use the Azure object, add the following line to your MediaScript:

```
#link "Azure.mdv"
```

Configuration:

Additionally, you will need to create a `local.properties_Azure` with the following key `Azure.ConnectionString= present`. The value for ConnectionString can be found in the Azure Storage Dashboard Access Keys tab under "Connection String". Either connection string can be used.

While it is a best practice to put this value in a separate file that can be secured, it will also work if you place this key-value pair in local.properties or local.properties_user.

If you prefer, you can manually break apart the connection string at the semicolons and place each value individually in the file, with "Azure." preceeding each key. (E.g., `Azure.DefaultEndpointsProtocol`, `Azure.AccountName`, `Azure.AccountKey`, and `Azure.EndpointSuffix`.)

Overview:

There are two ways to generate Azure SAS Blob Account URLS: `Azure.createBlobAccountSASFullURL()` and `Azure.createBlobAccountSASQueryURL()`. Blob service URLs cannot be generated at this time.

The difference between the two calls is the first generates a single-file fully-qualified URL while the second generates only the query string component which can be used with multiple files in the same storage account.

Methods:

```
string Azure.createBlobAccountSASFullURL(string RemotePath, int TimeToLive,
string Permissions)
```

`RemotePath` - The path to the file on the Azure remote server. This should not include the endpoint (i.e., https://xxxxx.blob.core.windows.net) since that will be acquired from the connection string local properties.

`TimeToLive` - The number of seconds the URL will be considered valid by the remote Azure server, measured in seconds. This time-to-live starts counting down from the moment the URL is created by this method, so leave enough time for any other script processing occurring after this call as well as the I/O operation itself to complete when choosing a value.

`Permissions` - A string with the characters indication which kind of operations will be allowed with this URL. Currently only "r", "w", and "rw" are supported. In the last case this allows the URL to be used for both Media.load() and Media.save() with the specified file.

The method returns a fully-qualified URL with the remote path inserted into the URL.

```
string Azure.createBlobAccountSASQueryString(int TimeToLive, string Permissions)
```

`TimeToLive` - The number of seconds the URL will be considered valid by the remote Azure server, measured in seconds. This time-to-live starts counting down from the moment the URL is created by this method, so leave enough time for any other script processing occurring after this call as well as the I/O operation itself to complete when choosing a value.

`Permissions` - A string with the characters indication which kind of operations will be allowed with this URL. Currently only "r", "w", and "rw" are supported. In the last case this allows the URL to be used for both Media.load() and Media.save() operations.

The method returns a query string that is suitable for appending to the end of a storage endpoint plus the path to the remote file.

```
string Azure.getEndPoint()
```

This method returns the cloud storage endpoint stored in the Connection String properties described at the top of this section of docs as a URL.

See the `Azure.createBlobAccountSASQueryString()` example to see how you can use it in your code.

Examples:

Example of loading and saving over the same file, then saving to a new file:

```
#link "Azure.mdv"
var url = Azure.createBlobAccountSASFullURL("testing/myfile.png", 900, "rw");
m.load(name @ url, detect @ true);
m.save(name @ url);
url = Azure.createBlobAccountSASFullURL("newfile.png", 900, "w");
m.save(name @ url);
```

Example of loading and saving saving to a new file:

```
#link "Azure.mdv"
var query = Azure.createBlobAccountSASQueryString(900, "rw");
var endPoint = Azure.getEndPoint();
m.load(name @ endPoint + "/testing/infile.png" + query, detect @ true);
m.save(name @ endPoint + "/testing/out.png" + query);
```

CHAPTER 5

MediaRich CORE Audio/Video 2

With the release of MediaRich CORE 4.0, AVCore 2 is the replacement for AVCore 1. Whereas AVCore 1 was written to emulate and expose the QuickTime API to MediaScript users, AVCore 2 is redesigned from the ground up to focus on high-speed transcoding with auto-assembly. Like AVCore 1, it works through MediaScript; however, the API is more similar to the AVCore 1 high-level API than to a low-level media SDK.

The MediaRich Audio/Video capabilities are exposed as MediaScript API, a library of MediaScript objects that extend the base capabilities of MediaRich. This chapter describes the use of that API.

MediaRich A/V capabilities are built into MediaRich, but are not enabled by default. To use the MediaRich A/V features in MediaRich, you must install a MediaRich license file (license.bin) that enables those features. A/V CORE Multi-threaded video processing functionality is an option, as well as GPU-accelerated processing using Intel Quicksync on SandyBridge and later processors. Standard MediaRich evaluation and production licenses do not enable A/V functionality unless requested. Contact an Equilibrium Sales Representative to obtain a license that enables the A/V features. Use the MediaRich CORE Administration utility to install any license file you receive from Equilibrium.

Chapter summary

MediaRich AVCore 2

MediaRich CORE 4.0 includes the AVCore 2 engine to power audio and video handling. It has been completely re-written as a 64-bit engine that allows it to take advantage of GPU acceleration and the ability to process large files, including 4K and 8k video.

AVCore 2 includes a broad range of functionality for automated audio/video processing:

- It reads all audio video formats supported by AVCore 1, **EXCEPT** FLI, FLC or any video containing paletted data.
- It writes most formats supported by AVCore 1 , plus native Flash Video, Ogg Theora, GXF, MTS, and VPX/webm.
- It is capable of auto-assembling several inputs with advanced normalization to make them all fit into a single output format.
- It provides a plug-in system for accessing “handlers”, which implement the low-level transcoding and processing functions for various media APIs.
- It provides a scalable growth path for the addition of GPU accelerators and other third-party encoder/decoders.

Optimized transcoding

While there are other utilities available that are capable of concatenating video and audio, they usually put strict limitations on the type of output container file and do not normalize the inputs, creating a “Franken-file" that consists of segments that run at different frame rates, contain variable video and audio formats, and are not capable of combining elements into a single frame. The AVCore 2 patented and patent pending technology is capable of doing all of this, while providing a fast transcoding environment during this process with the ability to modify each frame as it is output and encoded.

High-level APIs

With the use of handlers, developers do not need to learn low-level APIs because the built-in high-level API provides the abstraction and standardizes all functionality so a script using one handler will work with another handler. When a file is read, decoding handlers are selected by AVCore 2 depending on the file formats and codecs they support, and encoding handlers are specified by the user when they create a settings file to export video or audio to a particular file format and codec.

AVCore 2 Concepts

AVCore 2 is built on classes that provide the ability to read from file containers and then transcode the media contained in those files into new files, optionally appending and mixing them with other assets along the way. Here is a brief listing of constructs used in AVCore 2 to provide this reading and transcoding ability:

AVContainers

AVContainers provide access to the file being decoded, and can return information, such as the number and types of tracks the file contains and global metadata. Currently, they are read-only, so you cannot modify their data in-place. The only way to change the information inside a file is to export it to a new file and change the data as it is exported.

AVClips

AVClips provide access to the video and audio tracks inside the files. They are responsible for returning frames of video and chunks of audio as well as any track-specific metadata. AVClips can be the length of the entire track or can be a subrange of video or audio in a particular track.

VideoStills

VideoStills are a special kind of container in that they do not contain a file but contain the contents of a Media object along with an unlimited amount of audio silence. These are used to insert title cards into videos and generate silence for inputs that lack audio. The duration of the still or audio silence is determined when the AVClip is created from their AVContainer.

AVProcesses

AVProcesses are the transcoding engines. They are given an output file and settings file to use to determine the output format and the AVClips are added to them, which tells the exporter how to auto-assemble the final video or audio file. When all of the clips to be assembled are added, a single API call is used to begin transcoding. There are also optional callback capabilities in case effects, transitions, logos or other changes are needed to be applied as the frames are exported by the user.

AVRasters

AVRasters provide high-speed, abstract access to the decoder and encoder video frames. They currently cannot be manipulated directly except that they can be turned into MediaRich Media objects and back again. When they are Media objects, the user can use any of MediaRich's Media object functions to affect the look of the video frame before it goes out to the exporter, or the frames can be saved as still images or GIF animations.

AVAudioData

AVAudioData objects provide a pipeline for audio to pass from the decoder to the encoder. Currently they cannot be operated on, however future releases of AVCore 2 will provide the ability to adjust volume and mix audio.

Primary clips

The primary clip is the one that determines how other clips are conformed when they are exported into a single output video or audio file. While the settings file can potentially determine all of the formatting information for an output file, many settings files will be missing some information or use aspect ratio correction modes that require knowledge of the input. Because there may be several inputs, the notion of a “`primary`” input clip or input time was created so the information is from a single, most-important source.

Settings files

The settings files in AVCore 1 consisted of Quicktime atom containers or Windows Media .prx files. In AVCore 2, they are all text-based JSON files and the parameters in the settings files are mostly standardized across handlers (this enables easy programmatic changes, or simple editing manually). So "`bitrate`" for one handler means the same thing for another. It is possible to hand edit these files in a text editor if they need to be tweaked without having to recreate them in the settings generator.

AVCore 2 does not understand AVCore 1 settings files, so any custom settings you have created for the previous version must be recreated for AVCore 2. Most of the settings file that shipped with previous versions of MediaRich have already been recreated for AVCore 2.

Aspect ratio correction

In AVCore 1, aspect ratio correction was something that had to be done by the frame callback using `Media.scale()` with the "`constrain`" parameter set to true. In AVCore 2, aspect ratio correction happens automatically when the decoder's AVRaster is copied and converted into the encoder's AVRaster. The type of aspect ratio correction is determined by a parameter in the settings file.

See “Aspect Ratio Correction Modes” on page 258 to get a better understanding on how these work.

Normal and basic track assembly

In addition to an output track being either video or audio, a track can also be one of normal assembly or basic assembly. Normal assembly tracks are created when the first call to `AVProcess.addClip()` is done on that track. These tracks allow for overlapped clips where a frame callback could composite all of the frames occurring at that point in time in the output video into a single output frame. Basic assembly tracks, on the other hand, cannot overlap; one frame from one clip is passed into the callback at a time. These type of tracks are created when the first clip added to a track is added with `AVProcess.addBasicClip()`.

There is slightly less overhead in handling basic assembly tracks, but the primary reason they exist is to allow users that are already familiar with the AVCore 1 callbacks (which only allowed for one input frame to be processed without using tricks) to work in a similar workspace.

> ***Note:*** It is possible to create a normal assembly track with no clips that overlap; however the track will still be treated as a normal assembly track internally. It is not possible to mix these

track types; once a track is defined as normal only `AVProcess.addClip()` can be used on it, and once a track is defined as basic only `AVProcess.addBasicClip()` can be used on it.

AVCore API documentation

You will also find the full programmer A/V Core 2.0 API located in the MediaRich Server APIs. Additionally, all of the product demonstrations have a link to the API location.

Along with these resources, "AVCore 2 Best Practices " on page 266 provides useful information about working with the new A/V Core 2.0.

Compatibility Scripts for AVCore 1

MediaRich CORE ships with a compatibility layer to allow transcoding with existing scripts that relied on that feature in AVCore 1. The compatibility script replaces `QuicktimeMovie.ms` and provides most of the functionality of the old file, but it uses AVCore 2 and AVCore 2 settings files. If you do not require a frame callback to modify each frame as it is transcoded, it will run at full AVCore 2 speed. If a frame callback is required, you cannot take full advantage of the AVCore 2 speed because every frame must be converted into a Media object in order to remain compatible with `QuicktimeMovie.ms`. For more information about coding an optimal frame callback directly to AVCore 2, see "AVCore 2 Best Practices " on page 266.

`ExportMovie.ms` and `VideoStill.ms` are also replaced with AVCore 2 counterparts. The `SceneDetectFast.ms` script has not changed because it uses `QuicktimeMovie.ms` as an interface to AVCore 1 instead of using AVCore 1 directly, so it will work the same as before.

If you have an existing script that uses AVCore 1 low-level API calls, these will not work with AVCore 2. You must change them over to the AVCore 2 counterparts, if possible. The most likely calls are for retrieving video and audio metadata, as there was no high-level API in AVCore 1 for this purpose. AVCore 2 provides its own metadata API for enumerating, reading, and exporting metadata. For additional information, refer to the AVCore 2 API docs on the MediaRish server and to the `MetadataDemo.ms` script in the AVCore 2 Examples folder.

If you must use AVCore 1 for your project, the original scripts are renamed to "`QuicktimeMovie_Legacy.ms`", "`ExportMovie_Legacy.ms`", and "`VideoStill_Legacy.ms`". These scripts still use the AVCore 1 API; however, AVCore 1 is only available with 32-bit versions of MediaRich and not the now-standard 64-bit version.

Basic AV Transcoding

The following sections provide a simple example of transcoding two input files into a single output video. There is no error handling and the inputs and settings are presumed to be simple files located in the `MediaRichCore/Shared/Originals/Media` folder. Both of these files contain at least one video and audio track that run the entire length of the file.

Opening the Inputs

The first thing to do is open the files. The `AVContainer` object is used for this purpose:

```
var file1 = AVContainer("advertisement.mov");
var file2 = AVContainer("MyMovie.mp4");
```

It is NOT necessary for the containers to be the same file format; inputs can be just about any type of source. Next, we need to get at the video and audio tracks in each file:

```
var vclip1 = AVClip(file1, AVClip.kTrackTypeVideo, 1, 5);
var aclip1 = AVClip(file1, AVClip.kTrackTypeAudio, 1, 5);
var vclip2 = AVClip(file1, AVClip.kTrackTypeVideo);
var aclip2 = AVClip(file1, AVClip.kTrackTypeAudio);
```

Here we get access to a part of the first file's audio and video track and make it a clip (starting from one second in and lasting for five seconds), and for the second file we use the entire contents of its audio and video tracks. It does not matter what order we gain access to the tracks; we could have made the clips from file 2 first and then file 1. Remember these clips are abstractions; we aren't creating anything on the hard drive, only gaining access to the tracks within so we can grab info like video frames and audio samples.

Creating the Transcoding Processor

After opening the input files, create a processing object that will do the transcoding:

```
var processor = AVProcess("NewVideo.*", "MySettings.json");
```

This creates an output file named `NewVideo` upon execution of the transcoding process, using a settings file named `MySettings.json`.

The asterisk following the output filename causes the exporter to generate a proper filename extension based on the file format specified in the settings file. If you do not use an asterisk following the last "`.`", the auto-extension feature is not activated and the filename will be exactly what is specified. For example, if the "`MySettings.json`" file specified an output of MP4, the final filename is "`NewVideo.mp4`".

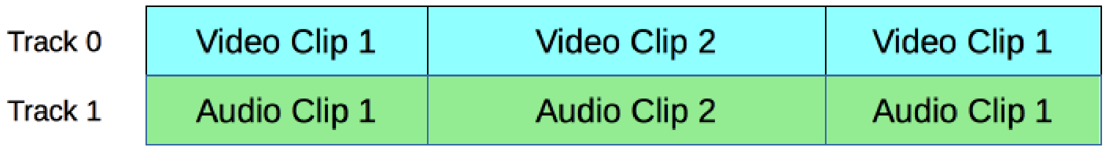
Note: If you need the final output filename before you exit your script, you can call `AVProcesses.getOutputPath()` after the transcode is complete to get that information.

Adding the Clips

After creating the transcoding processor, add the clips opened by the transcoding process to the processor:

```
processor.addClip(vclip1);
processor.addClip(vclip2);
processor.addClip(vclip1);
processor.addClip(aclip1);
processor.addClip(aclip2);
processor.addClip(aclip1);
```

The processor adds clips to matching tracks specified in the settings file, and if an appropriate track does not yet exist yet, it adds one. Because no starting time was specified in the `addClip()` call, all of these clips are appended to one another in their appropriate tracks. The following diagram illustrates how the output video is laid out when the transcoding is executed:



The track order is determined by the settings file; it would have been okay to add the audio clips first and then the video clips. The audio clips would have still been mapped to track 1 since that is how the output was laid out in the settings. If your settings specifies multiple audio or video tracks, there is a parameter you pass to `addClip()` (and `addBasicClip()`) to cause all subsequent clips to be added to the next matching track. (Refer to the API documentation.) It is also OK to have a settings file that specifies a particular track type and not add any of that kind of track. You will simply get an output file that only has a track with the type of clips that were added (audio or video).

Also notice that the first clip was added twice; it is totally permissible to add the same clips to an output more than once. The `addClip()` function returns the track number that was assigned to the added clips. This value can be used to adjust settings programmatically or extract information from the output track. (Refer to the AVCore 2 API documentation for more information.)

The basic example does not include any error checking and does not modify any of the settings, so it discards the returned track numbers.

Executing the Transcoding Process

Start the transcoding process with the following:

```
processor.execute();
```

The function will return when it is finished and the new output video is created in the specified path. If an error occurs during transcoding, the processor is invalidated and you must reinitialize it again with a fresh instance.

Error handling

All AVCore 2 errors throw exceptions (this is also the case with AVCore 1). So for normal error handling, you might need to add your code, test, and catch blocks. Uncaught errors will terminate the script and are logged to the `Scripterrors.log` file. If the script is run from an MRL, the web browser will also display the error message if it is not caught and handled.

Using AV Settings Files

AVCore 2 provides a built-in settings generator, which is a web-based interface built into MediaRich. This can be used simply as a way to create settings JSON files or it can be used as a user experience element in your own interfaces. With this utility, you can choose the handler, file format, codec, and various codec settings for each video and audio track you wish to export. For video tracks, you also specify how input media is reformatted to the output format, and whether aspect ratio correction should be applied, padding, and how additional assets auto-assembled into an exported video are conformed to the primary asset.

You can save these generated files as presets, or you can use existing presets located in the HotFolders default public access folder. Several compression setting files, as well as a PDF of settings, for popular formats are located in sample imaging examples area and the A/V Settings generation page here: http://localhost/mrm/AVCoreSettingsMaker/index.html.

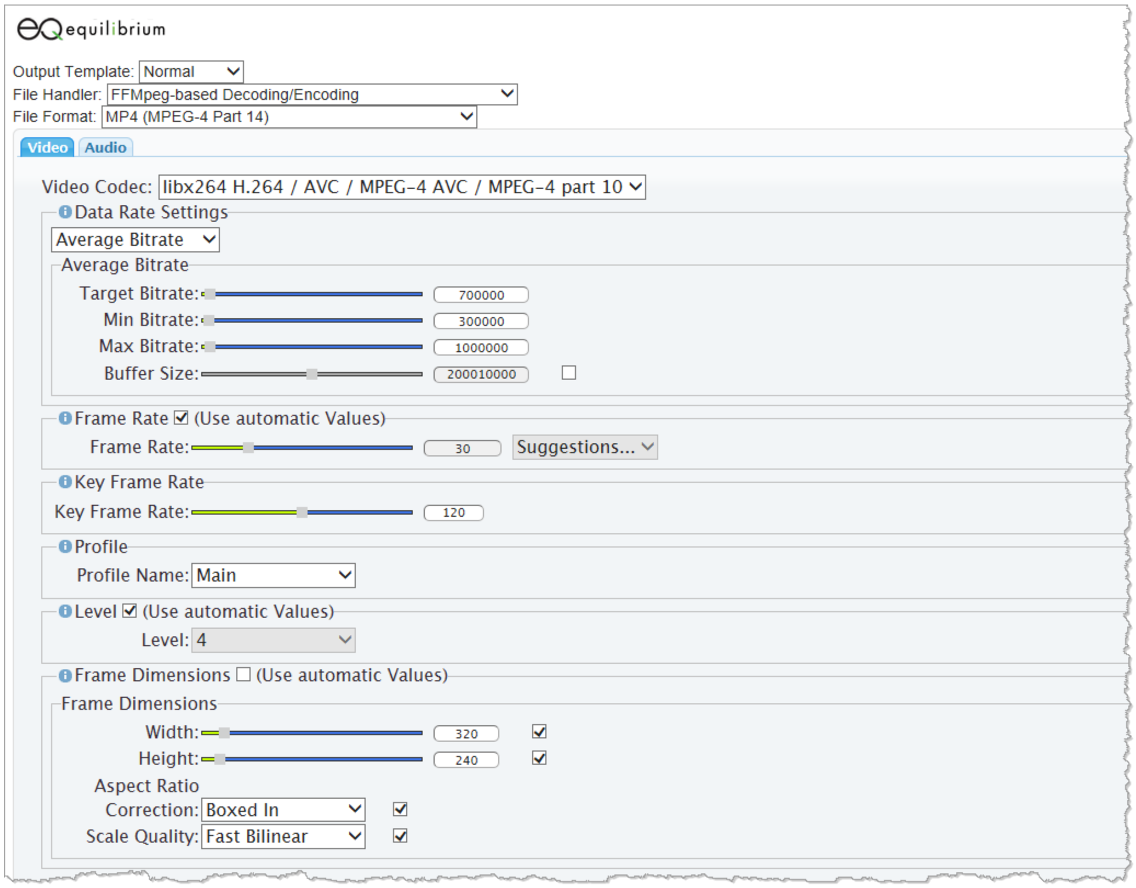
You can also download these files, but they must be made available to the MediaRich server. You can do this in the same way that you make any source asset available to MediaRich. For more information on where to place and how to access source asset files, see “File Systems” on page 39.

Accessing the AV Settings Utility

You can access the AV Settings utility from a browser using a standard URL:

```
http://<localhost or host IP>/avsettingsmaker/
```

Note: A 403 or 403.1 error can occur when trying to access this URL when logged onto the server itself. If so, your administrator must use IIS Manager to make changes to the avsettingsmaker permissions. This can typically be resolved by editing the Handler Mappings for Scripts to enable handlers that require script rights.


Output Template: Normal
File Handler: FFMpeg-based Decoding/Encoding
File Format: MP4 (MPEG-4 Part 14)

Video | Audio

Video Codec: libx264 H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10

Data Rate Settings
Average Bitrate
Average Bitrate
Target Bitrate: 700000
Min Bitrate: 300000
Max Bitrate: 1000000
Buffer Size: 200010000 ☐

Frame Rate ☑ (Use automatic Values)
Frame Rate: 30 Suggestions...

Key Frame Rate
Key Frame Rate: 120

Profile
Profile Name: Main

Level ☑ (Use automatic Values)
Level: 4

Frame Dimensions ☐ (Use automatic Values)
Frame Dimensions
Width: 320 ☑
Height: 240 ☑
Aspect Ratio
Correction: Boxed In ☑
Scale Quality: Fast Bilinear ☑

Creating a Settings File

The AV Settings utility generates text-based JSON files. The specified parameters in these files are standardized, for the most part, across handlers to enable easy programmatic changes, or simple manual editing in a text editor. For example, "`bitrate`" for one handler means the same thing for another.

This utility supports two file handlers:

- **FFMpeg-based Decoding/Encoding** - Use this file handler to do most audio-video tasks quickly and easily, including audio compression, audio/video format conversion, and image extraction. It handles both new and old video formats.
- **Intel QuickSync-based hardware accelerated encoding** - This file handler is designed for performance. If you need to do significant video transcoding with MediaRich, you should investigate getting hardware that has this capability so that MediaRich can take advantage of it. This produces a 2-6x performance boost over software encoding, depending on the size of the output frame (better performance for higher definition outputs, such as 720P and 1080P).

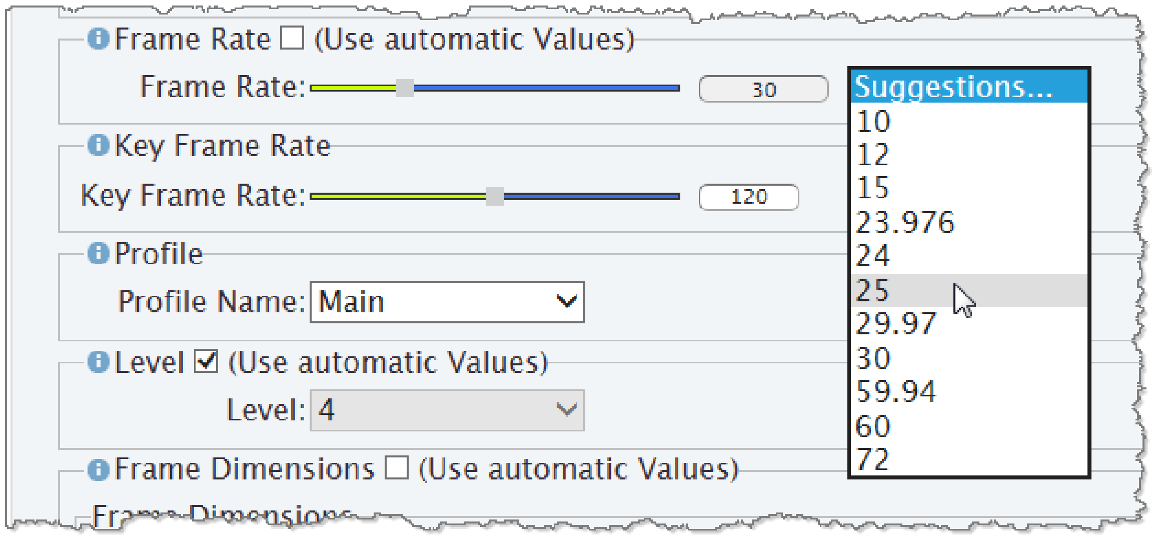
To create a settings files:

1. Open AV Settings utility.
2. Select the **Output Template**.

3. Select the **File Handler**.
4. Select the **File Format**.
5. On the **Video** tab, select the Codec used for video (choose any).

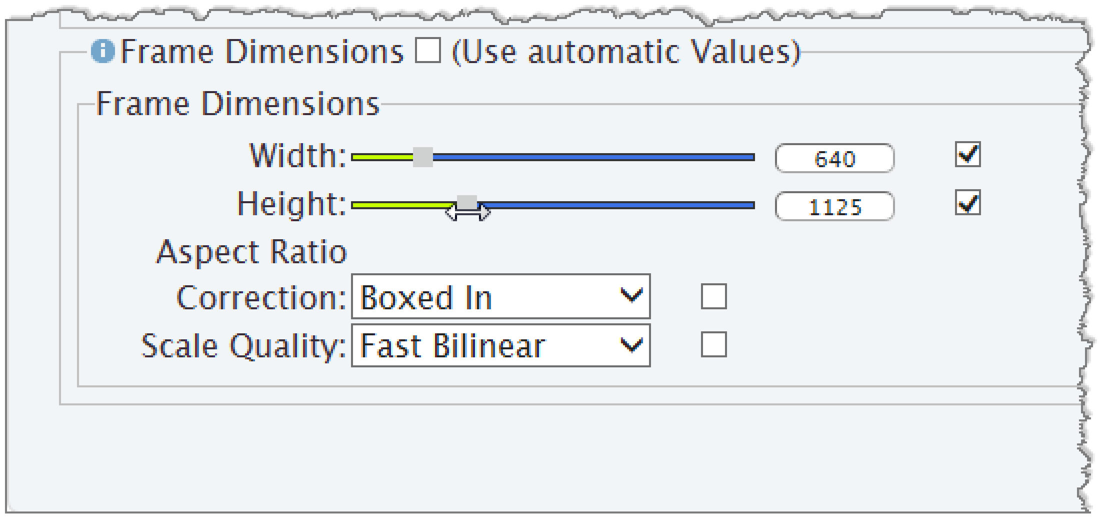
   Depending on the selected codec, you can specify how the input media is reformatted to the output format, and whether aspect ratio correction should be applied, padding, and how additional assets auto-assembled into an exported video are conformed to the primary asset.

   Frame Rate: This is a standard parameter, but the default value will differ depending on the codec. The (Use automatic values) option is selected by default, but you can clear the check box and use the Suggestions... menu to choose another value.



Frame Rate ☐ (Use automatic Values)
Frame Rate: 30
Key Frame Rate
Key Frame Rate: 120
Profile
Profile Name: Main
Level ☑ (Use automatic Values)
Level: 4
Frame Dimensions ☐ (Use automatic Values)

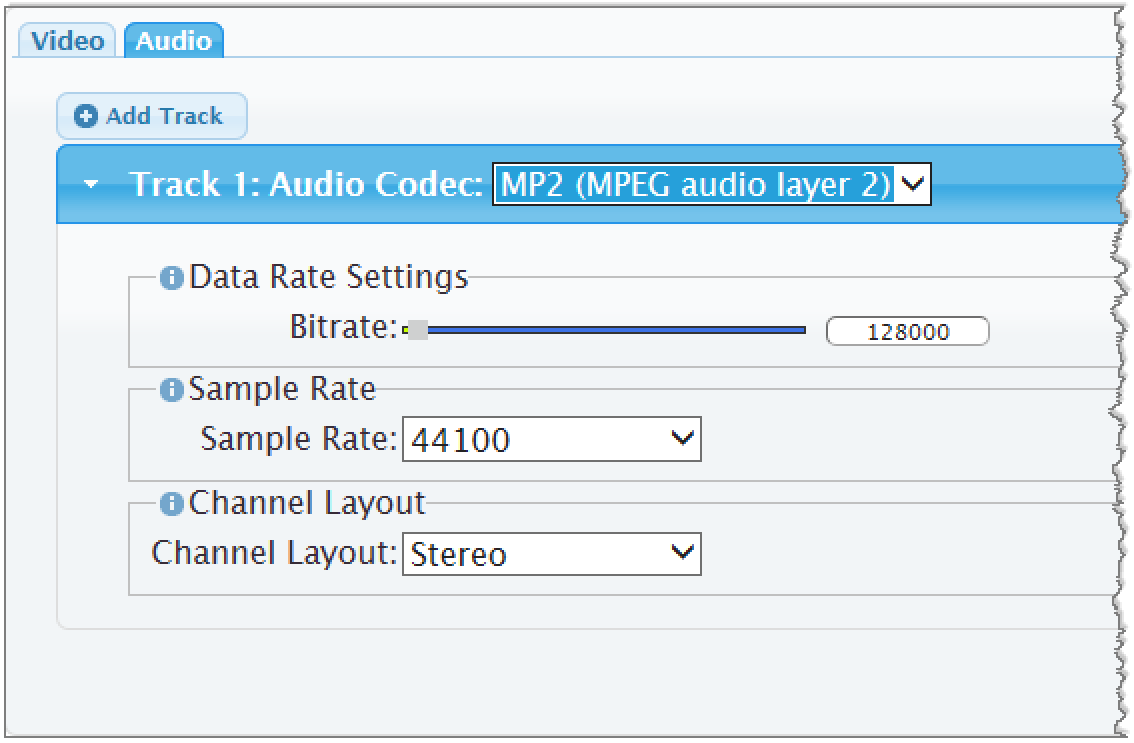
   Frame Dimensions: This is a standard parameter, but the default values will differ depending on the codec. The (Use automatic values) option is selected by default, but you can clear the check box and set each option as needed.



Frame Dimensions ☐ (Use automatic Values)
Frame Dimensions
Width: 640 ☑
Height: 1125 ☑
Aspect Ratio
Correction: Boxed In ☐
Scale Quality: Fast Bilinear ☐

6. Click the **Audio** tab and select the Codec used for each audio track (choose any).

   Some file format/video selections do not support a specific audio codec, or might support only one.



Track 1: Audio Codec: MP2 (MPEG audio layer 2)

Data Rate Settings

Bitrate: 128000

Sample Rate

Sample Rate: 44100

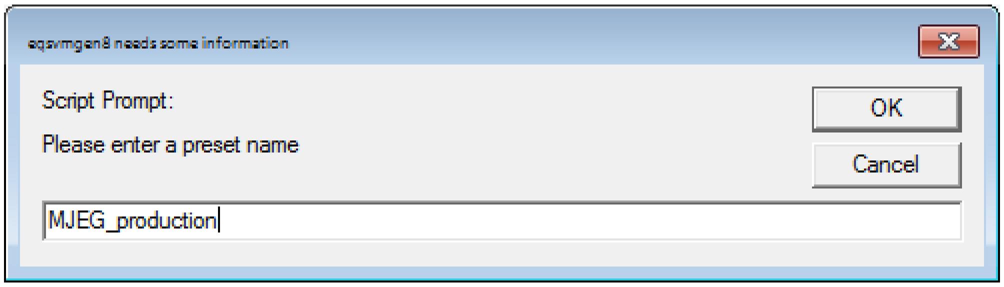
Channel Layout

Channel Layout: Stereo

Click Add Track to add an additional track and set the codec and parameters.

7. Save the file as a **Preset** or a **Download**.

   If you choose to save the file as a preset, you must enter a name for the preset file and click **OK**.



eqsvmgen8 needs some information

Script Prompt:

Please enter a preset name

MJEG_production

If you choose to download the file, your web browser download handles downloading and saving the file on your local system. To use the file, you will need to move the file to a folder accessible by your MediaScript on the MediaRich server.

Aspect Ratio Correction Modes

AVCore 2 provides fast, internal aspect ratio correction and normalization for video frames as they are passed from the source to the destination output frame. The aspect ratio correction mode is specified in the settings file, and can be configured by the AVCore 2 settings generator (`http://localhost/avsettingsmaker`). In the settings for the output file, you can specify the aspect ratio correction mode for each video track.

There are several aspect ratio correction modes. The following sections describe these modes and some include some diagrams to help illustrate and explain the results that they produce.

None

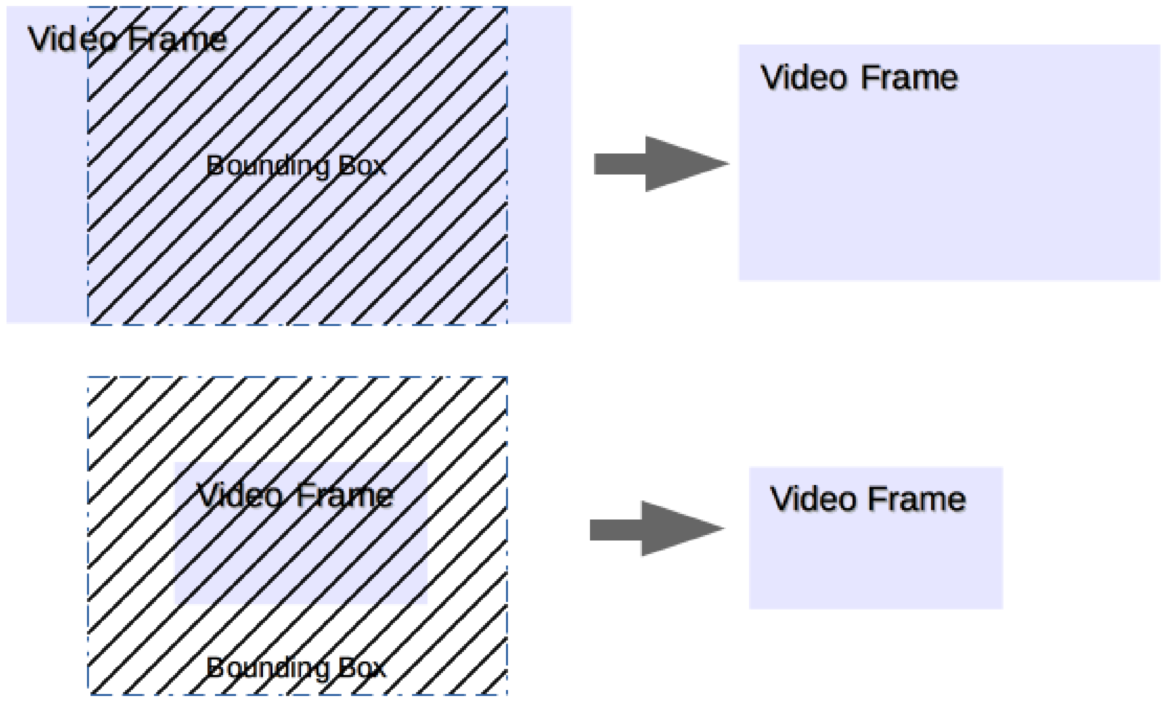
Specify None to disable aspect ratio correction. If both a width and height is specified in the settings file (or the output dimensions take on the size of the primary clip) the source frames are scaled to those dimensions, and stretched or squashed as needed without any regard to maintaining the original frame's aspect ratio. This will be true for all clips in the output video.

For example, if the primary clip is 1920×1080 and the second clip is 320×240 and there is no width or height specified in the settings, the second clip is scaled up to 1920×1080 (because the output takes on the size of the primary in this case) and is stretched.

Boxed In/All Boxed In

Use the Boxed In aspect ratio correction mode as the primary mode for transcoding wher possible. The width and height define a maximum bounding box size for the output video – anything larger than that is scaled down to fit inside the output dimensions with the aspect ratio kept intact. The primary video clip is left unscaled if it fits inside the box. This allows for the most efficient encoding size, since upscaling low-resolution videos doesn't improve quality and wastes bandwidth.

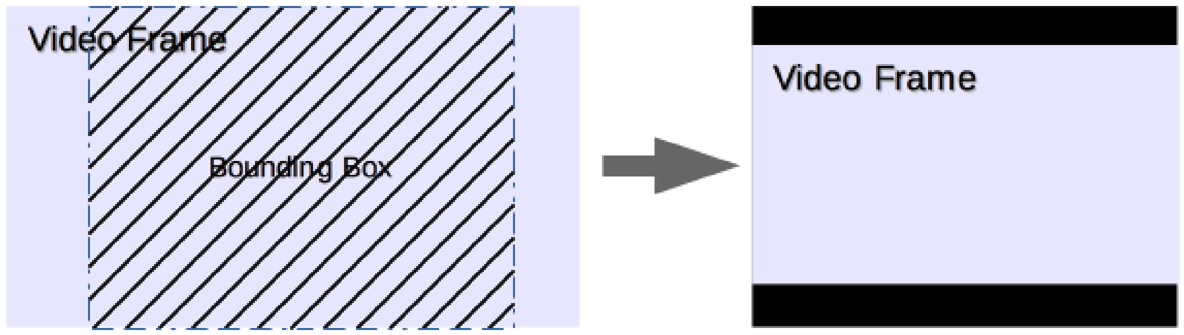
What is upscaled, however, are any non-primary clips. Because the output dimensions must remain constant (for many file formats, anyway), these secondary clips must conform to the primary clip format. So secondary clips are scaled to conform and padded as needed to fill out the frame to match that of the primary clip dimensions (using Pad Best Fit mode).



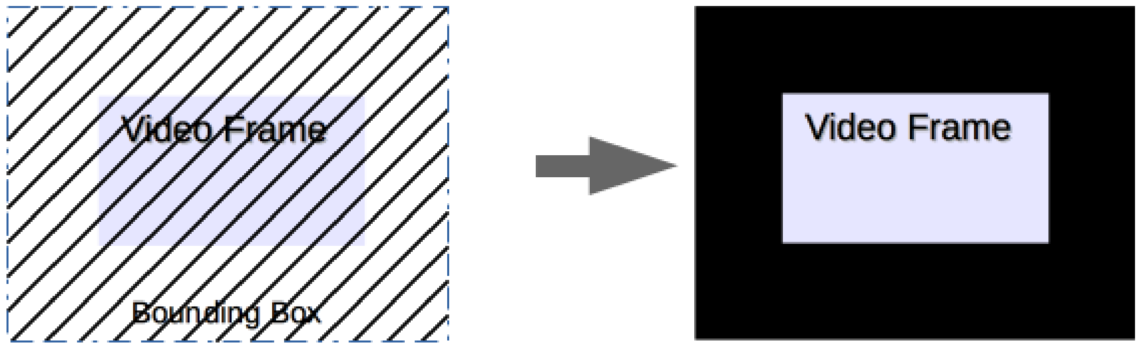
All Boxed In is the same as Boxed In, except it uses Pad Boxed In instead of Pad Best Fit for the secondary clips.

Pad Boxed In - Aspect Ratio Correction Mode

Pad Boxed In is similar to Boxed In, except primary clips that are smaller than the output dimensions have the extra space padded to fill the frame out to those dimensions.

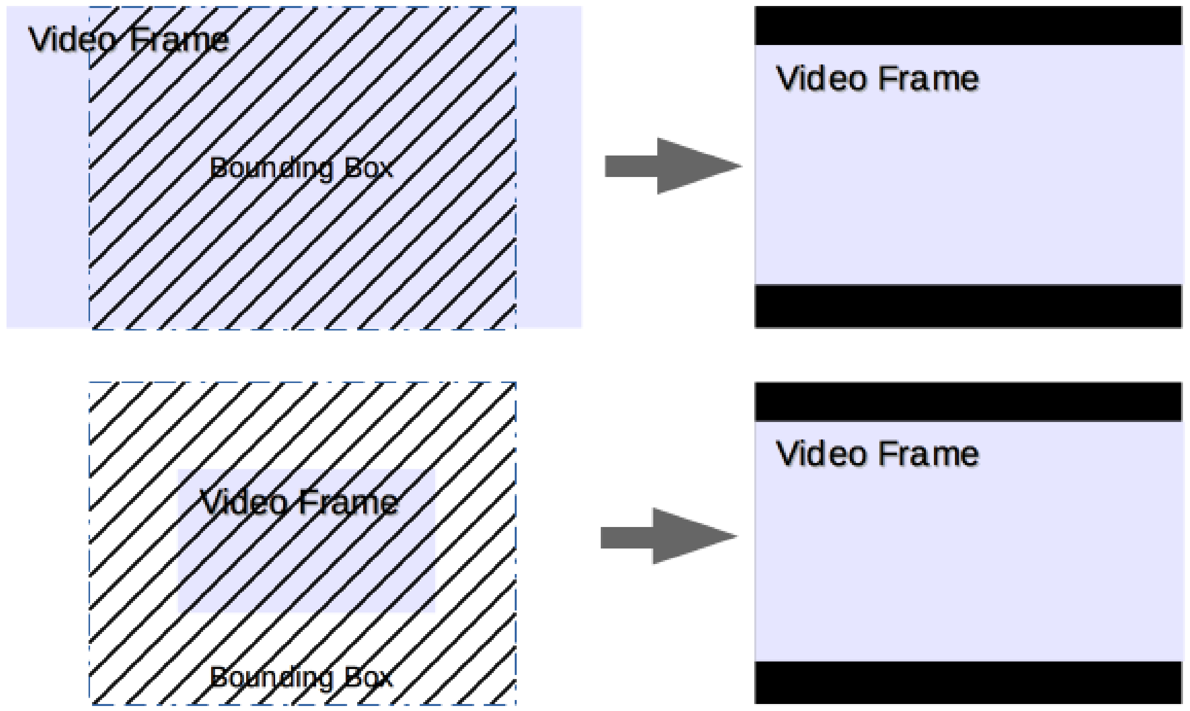


Also, if the primary clip is larger than the bounding box and is not the same aspect ratio as the bounding box, the primary clip is downscaled and padded to fill out the space to meet the bounding box dimensions and aspect ratio. All other clips in the sequence are also treated this way.



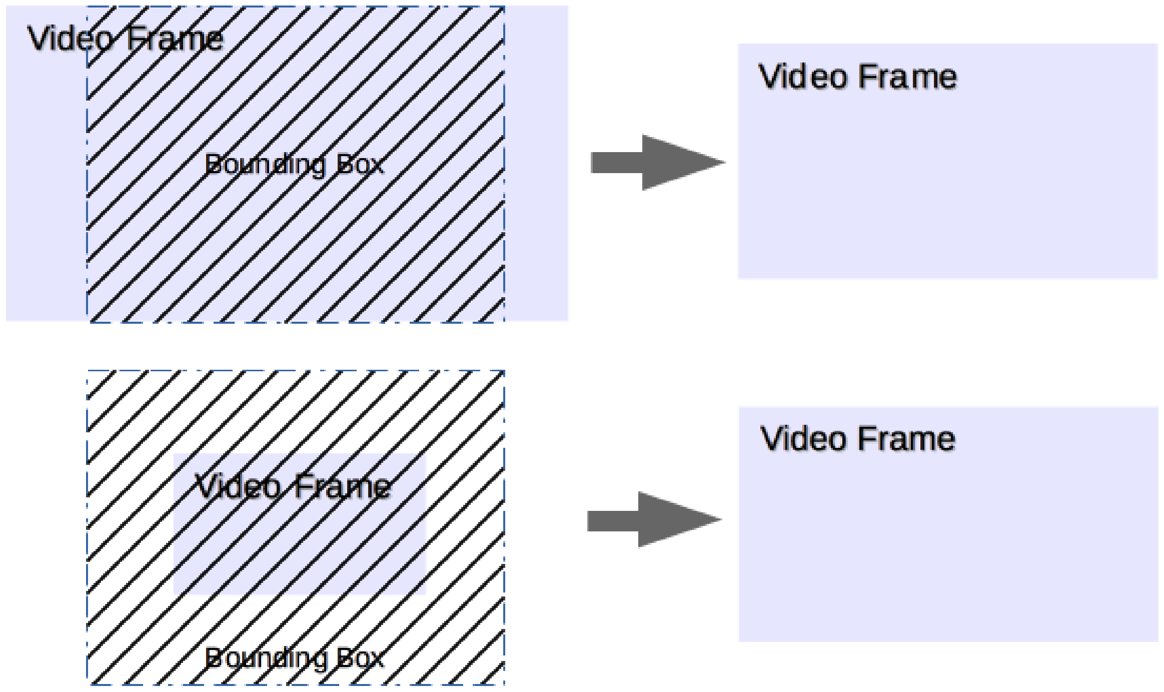
Pad Best Fit - Aspect Ratio Correction Mode

All clips are scaled up or down to fit inside the output frame dimensions, and if their aspect ratio is not the same as the output dimensions, padding is added to fill out the frame to those dimensions.



Best Fit - Aspect Ratio Correction Mode

The primary clip is scaled up or down to fit inside the output frame dimensions. No padding is added to this clip. All other clips are also scaled up or down but they are also padded so their sizes match that of the primary clip's new size.



AVCore 2 Callbacks

Each track in an output video is assigned a callback to handle the passing of video and audio to the output file. By default, AVCORE 2 uses an internal one written in C++ if one is not assigned to a track before `AVProcess.execute()` is called. This internal one is the fastest way to transcode videos, but provides no customization of the output.

If you need to operate on each frame of video, you can assign a custom callback instead of the internal default. The callback is a MediaScript function, similar to in AVCore 1. However, the parameters are different and, depending on the type of track assembly (normal or basic), there is a different callback for each type. Refer to the AVCore 2 API reference for the details on the callback parameters and required return values.

Note: Because `AVAudioData` has no operations, audio callbacks are of limited use at this time.

The following is a simple example that adds a callback that duplicates the functionality of the internal default video track callback for normal assembly:

```
function videoCallback(clips, outputInfo, destRast, &userData)
{
var srcRast = clips[0].clip.getVideoFrame(clips[0].reqTime, clips[0].duration);
if (srcRast)
{
destRast.convertFrom(srcRast);
return true;
}
return false;
}
```

This callback looks at the first clip in the stack of available clips at that particular time position in the output video and grabs a frame at the requested time from that clip. The AVCore 2 processor automatically calculates the correct requested time for each clip that needs processing and passes it as a member of the array entry for that clip. This is what the code that reads “clips[0].reqTime” is doing – it is using the calculated requested time and telling that AVClip to get a frame from that time position. The “`clips[0].duration`” is a return value. `AVClip.getVideoFrame()` returns the frame's duration in the second parameter and we are storing it back in the clip array under the required member name “duration”. After the callback has completed, all of the returned durations are examined and the shortest one is used to increment the time position in the output video, if the settings file did not specify a frame rate and the encoder is capable of variable frame rates. (If a fixed frame rate is used or is required by the encoder, the output time is simply incremented by the output frame rate duration and the duration info is discarded.)

It is a best practice to check to make sure `AVClip.getVideoFrame()` returns an actual frame (or chunk of audio in the case of audio clips). This is because in some cases the clip representing the source track may not be able to return a frame due to an error in the file or some other problem. In such cases, this callback simply aborts the transcode by returning false from the function. The callback has final say over when the output video ends – you could terminate the video short if you wished, so only return false if you really want the video to end at that time.

The source frame is copied and converted into the destination output video's frame by the code that reads “`destRast.convertFrom(srcRast)`”. `DestRast` is the exporter's output raster, and whatever is copied into it is first converted to its width, height, and pixel format. If an aspect ratio correct mode was specified in the settings file, black padding is also added if needed to make the input conform.

The following is an example of how a basic assembly track's callback might look like:

```
function videoCallbackBasic(clip, clipNum, reqTime, remTime, inFrameNum,
outputInfo, destRast, &userData)
{
var duration;
var srcRast = clip.getVideoFrame(reqTime, duration);
if (srcRast)
{
destRast.convertFrom(srcRast);
return duration;
}
return -1;
}
```

The big difference with basic callbacks is the clip is not an array – it is a single clip (since only one will be passed in at a time due to everything being an append) and all of the other info that was normally in each entry in the clip array is now spread out across the parameters. (Normal clip callbacks all have this info as well, it's simply more neatly contained.) The other difference is basic callbacks return the duration of the frame of the clip with the function, and return -1 to terminate the transcoding process.

In both types of callbacks, the last parameter is a user data object. The MediaScript program can pass any type of variable or object to the callbacks, and this is entirely for the user's use. The user data parameter is assigned on a per-track basis, just like the callback is, and in fact is assigned to the track at the same time as the callback:

```
processor.setTrackCallback(vidOutTrackNum, videoCallback, myUserData);
```

In this case if this was an extension of the example transcoding script in the previous section we would have had to get the video track number from `addClip()` and store it in the variable "`vidOutTrackNum`" in order to tell the processor which track was being assigned a custom callback. The variable named "`myUserData`" is an example of passing a variable to the callback. If you do need a custom callback but don't need to pass user data, it is recommended to leave off the user data parameter to this function call – AVProcess will automatically pass in an undefined type to the callback for the user data for you.

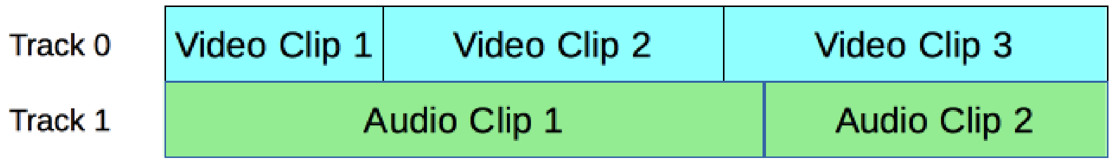
Working with AVClips

Video clips and audio clips are completely separate entities – a clip created from a video track knows nothing about any of the other tracks that were contained in its parent file. For this reason, the default duration of AVClips is the duration of the entire parent container. This means the duration of both the audio and video tracks from the same container file will be the same unless a start time or duration is specified when the AVClip is constructed.

Because AVClips automatically fill in missing audio or video until their duration is reached, this keeps any clips that are laid out in an output movie in sync. Note that sources with damaged audio information may still lose sync if the damage is somewhere besides the start or end of the audio track. There is currently no time stretching available in AVCore 2 to fix these damaged audio tracks. Video tracks with missing frames will have their previous frame extended to fill in the missing ones, however.

It is not allowed to lay out tracks in an AVProcess that have gaps. If you call `AVProcess.addClip()` and add a clip at a time position that creates a null space in the track, you will get an error when `AVProcess.execute()` is called. Clips can overlap and butt up end-to-end, but they cannot be placed sparsely. If you need empty space in a track, create an empty VideoStill and use it to fill in the gap you want to have in the audio or video track.

It is possible to create voice over or soundtracks that replace existing audio from various video file inputs. The video tracks are assembled as they normally would, but the audio track would come from a different container source than the video track clips. That source can have clips that do not match with the ends of the video clips. If it is required that both audio and video tracks end at exactly the same time, the clips on one of the output tracks can be lengthened or shortened by specifying the duration of the source clip. For audio clips, AVClip will lengthen it with silence or truncate it as needed so it ends at the specified duration. For video clips, the last frame in the clip is repeated until the duration is met. The following diagram illustrates how an output video might look using this technique:



AVCore 2 Metadata

AVCore 2 provides access to video and audio metadata with a few API functions. This is in addition to the standard Exif, XMP, and IPTC metadata support provided by MediaRich. (For more information about the general metadata support, see CHAPTER 10 , “MediaRich Metadata Support” on page 326.)

You can read and store metadata at the container/file level and at the track level. Each metadata value is kept in a named key, and all keys are string values. Metadata values can be either strings or binary data; however, the two handlers included in MediaRich CORE 4.0.0 only use string metadata at this time.

Note: There is no commonality between metadata keys for different file formats and the codecs assigned to each track. For example, some containers might support metadata stored under the key name “title”, but others might not or it could be a different key. There is currently no automatic mapping system for metadata being read in from one container or codec and stored in another. Such a mapping system could be created in MediaScript without too much trouble; the main part of the work being to learn and research the kinds of metadata the containers and tracks for each format planned for use can hold.

To see what metadata is available in an opened file or track/clip, use the `getMetaKeys()` function on the `AVContainer` to enumerate available data at the file level, or use the same function on an `AVClip` to get it at the track level. This function returns a comma-separated list of available key names, which can be split() and then all individual values can be read using either `AVContainer.getMetadata()` or `AVClip.getMetadata()`. Refer to the `MetadataDemo.ms` script for an example on passing any available metadata from a source video to a new output video.

AVCore 2 Best Practices

The following sections provide helpful information about working with AVCore 2.

Pixel formats

While AVCore 2 supports several different pixel formats, paletted video frames are **not** supported at this time. For this reason, these types of videos must be re-saved as RGB or YUV before processing with AVCore 2.

Avoid using Media objects in the frame callback, if possible

At this time, Media objects are not optimized for video transcoding and will slow down the transcode process. One of the major AVCore 2 optimizations over AVCore 1 is the ability to avoid raster pixel format conversions whenever possible. If all inputs are YUV420P and the output is YUV420P, the frames will pass through without requiring conversion to something else (although there could be scaling and padding if needed.) With AVCore 1 (and when using Media objects in AVCore 2) the rasters are converted to RGB pixel data, which can cause a slight loss in quality as well as incurring the overhead of having to go from YUV->RGB->YUV for each frame or sub-element in a scene.

Since `AVRasters` do not currently support compositing, drawing text, or any other operations other than copy-convert—you will still need to use a Media object during some points of the transcode if you need to modify the frames in some way. The best way to handle this is to write the callback to only convert to a Media object when needed, and at every other time position in the video you should pass back the source `AVRaster` from the clip directly to the destination encoder raster. Examples of this can be seen in the `FadeDemo.ms` and `DissolveDemo.ms` sample files, which only use the Media object when the transitions need to be applied.

In some cases, it will not be possible to do this. For example, overlaying a logo over an entire video will require every frame be turned into a Media object, composited, and then converted back into the destination raster's format. If you do wish to speed up transcodes for logo overlay, consider only displaying the logo for part of the video, and have it fade in and out from time to time.

Avoid using Media.scale() if possible

The `scale()` function in the Media object can be a major bottleneck in video transcodes, and because there is built-in fast scaling and padding when the frame is passed to the encoder, it is often not necessary. In AVCore 1, there was no built in aspect ratio correction in the QuickTime encoder, so it had to be done by the Media object for every frame. In AVCore 2, this is not the case and you will see a slowdown if you use your existing AVCore 1 callback functions 1 without making changes.

However, there are still cases where `Media.scale()` will be required. Because the padding is applied after the frame has passed from control of the callback to the encoder, overlaying logos, text, or dissolves will require `Media.scale()` be used during those times to normalize the frame for the elements being applied. The base frame will need to be scaled to the output dimensions (helpfully provided to the callback; see the examples and API ref) with constrain set to true, and then you can apply your elements to that base frame and pass it to the encoder's destination frame. This is only required if you are unsure of the aspect ratio of all your inputs – if you know they all have the same aspect ratio, then scaling with `Media.scale()` is not necessary since any padding (if any) will be the same for all of them. The only thing to do is calculate the correct position of the object that needs to be overlaid since the inputs width and height could still change.

Prescale composited elements, if possible

In some cases you might be able to prescale the elements you are planning to composite onto a base frame, if you can gather enough information about the clips being assembled. This will avoid some of the `Media.scale()` calls done in the callback, but it also will require more memory to keep all of these pre-scaled assets around as Media objects.

Don't use VideoStills for overlay elements

The main purpose of the `VideoStill` is to insert something into a sequence that isn't going to require compositing via the Media object. If you need a static logo over every frame, it is more efficient to simply use the original Media object that contains the image instead of converting it to a `VideoStill` and then back into a Media object in the callback for the compositing. `VideoStills` are meant for title cards or other still images that need to be inserted into a sequence and will be displayed on their own (or are the base element) since they exist to emulate a video clip.

Check the inputs before operating on them

The `AVContainer()` and `AVProcess()` classes have several member functions that you can use to query the contents of the input file and settings file, respectively. It is better to use these methods instead of trying to add clips for tracks that don't exist in the input or output and expecting to catch the errors that result after the fact.

For `AVContainer`, the `hasVideo()` and `hasAudio()` are very useful and provide a quick, easy way to determine if an input file has any video or audio tracks in it at all. You can then skip executing all the portions of your code that relate to these types of tracks. Similarly, `AVProcess` also supports these methods, and can quickly tell you if the settings file has any video or audio tracks defined for the output file. Refer to the AVCore 2 examples to see one way these concepts can be implemented.

CHAPTER 6

MediaRich CORE Audio/Video 1.1 (Legacy)

MediaRich Audio/Video version 1.1 is implemented on top of the Apple QuickTime technology, and its capabilities are closely tied to capabilities of QuickTime. In addition, the PC version enhances the QuickTime functionality by allowing it to read and write Windows Media (WMV/WMA) files.

Important: As the QuickTime professional transcoding capability is deprecated, the MediaRich CORE A/V 1.1 is also deprecated. We highly recommend that you do NOT utilize the MediaRich CORE 1.1 API unless absolutely necessary for a specific operation. **A/V CORE 1.1 is 32-bit only**, so you will need to retrieve the 32-bit MediaRich 4.0 before proceeding with any A/V CORE 1.1 usage.

The MediaRich Audio/Video 1.1 capabilities are exposed as MediaScript API, a library of MediaScript objects that extend the base capabilities of MediaRich.

Chapter summary

A/V CORE 1.1 Overview

MediaRich A/V can be used to process and create audio and video files in a variety of formats. To simplify explanations, all such files are referred to as “movie” files.

MediaRich A/V Core 1.1 is exposed to the MediaScript programmer as a number of objects (classes) and functions. These entities become available within a script by including the “QuicktimeMovie.ms” MediaScript file.

The primary object used to process movie files in MediaRich CORE A/V 1.1 is the “QuicktimeMovie” object. A QuicktimeMovie object represents a single movie file on which to perform operations. Using this object, you can query the movie metadata (width, height, frame rate, etc.), extract individual frames, and process and recompress the movie.

The ExportMovie function is called when multiple movies are to be combined to produce a new movie file. The calling script first instantiates a QuicktimeMovie object for each of the source movies, and then passes those objects, along with additional parameters, to the ExportMovie function to produce the derivative movie.

In version 1.1, ExportMovie supports overlay of alternate audio track, replacing the existing audio in the main inputs

A/V Core 1.1 Licensing

The MediaRich A/V 1.1 capabilities are built into MediaRich, but are not enabled by default. To use the A/V 1.1 features in MediaRich, you must install a MediaRich license file (license.bin) that enables those features. Standard MediaRich evaluation and production licenses do not enable A/V functionality. Contact an Equilibrium Sales Representative to obtain a license that enables the A/V features. Use the MediaRich CORE Administration utility to install any license file you receive from Equilibrium.

Adding QuickTime Support

In order to use any of the QuickTime video features on the Windows platform, you must ensure that QuickTime is installed. You can download the QuickTime player for Windows at http://www.apple.com/quicktime/download/. Run the setup file to install QuickTime. To work with Flash files, you must also have a Flash encoder QuickTime plug-in installed.

QuickTime wrapper: low-level audio/video API

If you plan to use the lower-level QuickTime classes and methods, these are documented in AVLowLevelDocs.html, which is located in the MediaRich All Media Server/MediaRich Documentation directory in your MediaRich server installation.

Using MediaScript to Access A/V 1.1 Objects

To access the MediaRich A/V 1.1 API, a script must first include the QuicktimeMovie.ms file. This is done with the following line of code:

```
#include “sys:/QuicktimeMovie.ms”
```

Error handling

Most of the APIs methods and functions throw an exception if an error occurs. If the exception is not caught by the script, the script will stop executing and an error will be returned to the client application.

Units of time values

Most time values passed to and returned by the API methods and functions are in seconds. Some functions have versions that accept QuickTime timescale units. These functions all end with the string "ByTimescale".

Instantiating a QuicktimeMovie object

To begin working with an existing movie file, a script instantiates a QuicktimeMovie object, passing the path to the QuickTime file as a parameter to the QuicktimeMovie constructor function.

new QuicktimeMovie(movieFile) constructor

The `QuicktimeMovie(movieFile)` constructor creates a new QuicktimeMovie object.

Syntax

```
var movie = new QuicktimeMovie(“MyMovies/TVSpot1.mpg”);
```

Returns

A QuicktimeMovie object representing the specified movie file.

If the specified movie file does not exist or is not readable, an exception is thrown.

SMPTE Time Code Support

A/V Core 1.1 provides SMPTE Time code support with Quicktime 7.6+ installed.

If your media source already has a SMPTE time code track stored in it (if it comes from a professional piece of capture equipment, this should be the case), all you need to call is the following:

```
SMPTETime = QTMovie.timeToSMPTE(<time in seconds>)
```

And the corresponding SMPTE time is returned in an object that contains the following fields: hours, minutes, seconds, frames, and text. The text field contains an actual SMPTE-formatted text string with the time in a printable format.

If the movie does not have an SMPTE track, you can call `QTMovie.addTimeCode(...)` to add one. This method takes the following parameters:

- `UseDropFrame` - for color NTSC, 2 frame #'s must be skipped every minute except at minutes divisible by 10. This is the standard because NTSC isn't really 30 fps so set this param to “`true`” to enable this.
- `TimeScale` - Frame rate of the source material. 2400 for film, 2500 for PAL/SECAM, 3000 for B&W US video, 2997 for NTSC color video.
- `FrameDuration` - set to 100. (You must use other TimeScale values above if you set this to something other than 100.)
- `NumFrames` - Number of frames that make up one (1) second of video. Use 24, 25, 30, and 30 respectively for each standard listed above.
- `Hours, Minutes, Seconds, Frames` - Each of these parameters gives an offset to the starting time that will correspond to the first frame of video in the movie. (Defaults to zeros for all of them (00:00:00;0) if not specified.)
- `Visible` - If true, a visible display of SMPTE time is shown during movie playback in QTPlayer.
- `Track` - Normally the first video track is bound to the SMPTE track. This lets you specify another track.

Important: In order for this to update the file, the movie must be opened with writing enabled (`new QTMovie(name, Quicktime.newMovieActive, Quicktime.fsRdWrPerm)`). Otherwise, the changes will be lost. You also must call `QTMovie.updateResource()` to write the changes out.

- `QTMedia.getSample()` function allows return of raw samples from a track's media. This allows for text tracks to be returned.

The following example code shows how to display all the text in a movie containing a text track:

```
function textTest(inname)
{
var movie = new QTMovie(inname);
var track = movie.getIndTrackType(1, Quicktime.TextMediaType,
Quicktime.movieTrackMediaType);
var media = track.getMedia();
var curtime = 0;
while (true)
{
var duration;
var mystr = media.getSample(Quicktime.TextMediaType,
curtime, curtime, duration);
if (!mystr)
break;
print("Found text sample at "+curtime+"\n");
print(">"+mystr+"<\n");
curtime += duration;
}
movie.dispose();
}
```

Note: `Movie.getNextInterestingTime()` does not return the correct times with text tracks, so the time and duration from `getSample()` must be used instead.

Using FFMpeg Objects

This object provides a wrapper for the FFMpeg tool. It simply executes that tool, passing it input and output paths and a set of user-supplied options. This enables FFMpeg export to be used for proxy video and other transcodes as a third-party expansion.

For example, Flash Video transcodes can be provided in MediaRich for FLV proxy generation, or previews without requiring the installation of a special FLV plug-in for QuickTime. It does provide this functionality under the umbrella of the MediaRich processing architecture, providing fault tolerance and scalable deployment.

Note: This is not interoperable with the frame handling system in the A/V Core 1.1.

To use this object, the ffmpeg.exe binary, along with its supporting shared library, must be installed in the Bin/MediaEngine directory.

Use of the FFMpeg link library looks like the following:

```
var ffmpeg = new FFMpeg();
try
{
ffmpeg.execute(
"avikitex:/Media/iMac_Dance_2997.mpg", "avikitex:/Out/_MPEG2_NTSC.mpg",
"-target xxntsc-dvd -b 3000k -ab 224k -sws_flags experimental");
}
catch (ex) {
throw ex + "\n" + ffmpeg.getLastError();
}
```

The FFMpeg object has two methods:

`execute` - This method takes an input path, an output path, and a string containing a list of ffmpeg options that are passed through to the ffmpeg tool.

`getLastError` - If the execute method throws an error, this method can be called to get the error message that was returned by the ffmpeg tool.

Querying Movie Metadata

After the AVCore 1.1 `QuicktimeMovie` object is instantiated, that object can be used to obtain information about the movie file that it represents. The following methods of the `QuicktimeMovie` object are used to access the movie characteristics.

<movie>.getWidth()

The AVCore 1.1 `<movie>.getWidth()` method returns the width of the movie, in pixels.

Syntax

```
var width = movie.getWidth();
```

Returns

An integer value indicating the width of the movie.

<movie>.getHeight()

The AVCore 1.1 `<movie>.getHeight()` method returns the height of the movie, in pixels.

Syntax

```
var height = movie.getWidth();
```

Returns

An integer value indicating the height of the movie.

<movie>.getFileName()

The AVCore 1.1 `<movie>.getFileName()` method returns the base file name of the movie file.

Syntax

```
var name = movie.getFileName();
```

Returns

A string containing the movie file name.

<movie>.getFilePath()

The AVCore 1.1 `<movie>.getFilePath()` method returns the full path of the movie file.

Syntax

```
var path = movie.getFilePath();
```

Returns

A string containing the movie file path.

<movie>.getFrameCount()

The AVCore 1.1 `<movie>.getFrameCount()` method returns the number of video frames in the movie.

Syntax

```
var frames= movie.getFrameCount();
```

Returns

An integer indicating the number of frames in the movie.

<movie>.getDuration()

The AVCore 1.1 `<movie>.getDuration()` method returns the length of the movie.

Syntax

```
var duration = movie.getDuration();
```

Returns

A floating point value indicating the length of the movie, in seconds.

<movie>.getNumChannels()

The AVCore 1.1 `<movie>.getNumChannels()` method returns the number of channels of the first audio track in a movie.

Syntax

```
var numChannels = movie.getNumChannels();
```

Returns

An integer value indicating the number of audio channels.

<movie>.getSampleFormat()

The AVCore 1.1 `<movie>.getSampleFormat()` method returns the size of each audio sample of the first audio track in a movie.

Syntax

```
var formatInfo = movie.getSampleFormat();
```

Returns

An object that contains the following member: size. This member contains an integer indicating the audio sample size in bits.

<movie>.getSampleRate()

The AVCore 1.1 `<movie>.getSampleRate()` method returns the sample rate of the first audio track in a movie.

Syntax

```
var sampleRate = movie.getSampleRate();
```

Returns

An integer value indicating the movie audio sample rate.

Extracting Frames from a Movie

After the AVCore 1.1 `QuicktimeMovie` object is instantiated, you can extract any number of individual frames from the underlying movie file as MediaScript Media objects.

<movie>.getFrameAtTime(time)

The AVCore 1.1 `<movie>.getFrameAtTime(time)` method extracts a frame from the movie.

Syntax

```
var frame = movie.getFrameAtTime(2.5);
```

Parameters

`time` - the time, in seconds, of the frame that should be extracted.

Returns

A Media object containing the frame at the indicated time. See “Media Object” on page 68 for information about media operations and saving the contents of the returned Media object.

<movie>.saveFramesAtTimes(times, outPathFmt, frameWidth)

The AVCore 1.1 `<movie>.saveFramesAtTimes(times, outPathFmt, frameWidth)` method extracts a series of frames from the movie and saves them to a set of numbered files, optionally scaling each frame before saving it.

Syntax

```
var frame = movie.getFrameAtTime(times,“thumbnails/thumb_%d.tif”, 128);
```

Parameters

`times` - an array of floating point values representing the times of the frames to be extracted from the movie.

`outPathFmt` - a printf style format string that specifies the path and names of the resulting output files. This string should contain a single instance of the format specifier ‘%d’ to indicate where the frame number should be expanded into the output file name.

`frameWidth` - an optional parameter that indicates the desired width of the frames to be saved. If this parameter is not specified or is 0, then the frames will not be scaled and will therefore be the same size as the movie. If a non-zero value is specified, then each frame will be scaled proportionally to have the indicated width.

Returns

This method returns no objects.

Using ExportMovie to Create New Movies

Use the ExportMovie function to create new movie files. This function takes a list of input sources as well as parameters that describe the operations to be performed and compression settings to be used to create the new movie file.

ExportMovie is a complex function. To supplement this information, there are sections that provide detailed information about its parameters.

ExportMovie(inputs, outputPath, settings, processObject)

The AVCore 1.1 `ExportMovie(inputs, outputPath, settings, processObject)` method creates a new movie file from input sources and parameters.

Syntax

```
ExportMovie(inputs, outputPath, settings, proc, audioInputs);
```

Parameters

`inputs` - a single input source, or an array of input sources.

`outputPath` - the path of the movie file to be created.

`settings` - the format and compression settings to be used to compress the movie.

`processObject` - an optional parameter that describes the image processing operations to be performed on each frame of the movie.

`audioInputs` - an optional parameter that causes the audio to be replaced with another set of audio.

Returns

This method returns no objects.

ExportMovie: Inputs Array

The “inputs” parameter for `ExportMovie` consists of an array of one or more input sources to be used to construct the new movie file. Input sources can be any combination of `QuicktimeMovie` and `VideoStill` objects. As a convenience, a single `QuicktimeMovie` or `VideoStill` object, or the path to a movie file can be passed in the “inputs” parameter. In these cases, that single object or file is used as the only input source for the export operation.

The content of the new movie consists of each input source, appended one after the other, in the order that they appear in the input array. Multiple segments from a single input movie file can be specified by creating multiple `QuicktimeMovie` objects that refer to that same movie file.

Each `QuicktimeMovie` object in the “inputs” list represents a segment of a movie file. By default, the entire length of a source movie is added to the destination movie. By calling the `setSegment()` method for a `QuicktimeMovie` object, you can add an arbitrary portion of the underlying movie to the output.

Example

To help illustrate the use of the `inputs` parameter, the following example creates a new movie consisting of the first five seconds of a source movie, followed by a one-second blank blue frame, followed by a complete second source movie, and ending with a copyright image displayed for five seconds.

```
var inputs = new Array();
inputs[0] = new QuicktimeMovie(“InputMovie1.mov”);
inputs[0].setSegment(0.0, 5.0);
inputs[1] = new VideoStill(inputs[0].getWidth(), inputs[0].getHeight(), 0x0000ff,
1.0);
inputs[2] = new QuicktimeMovie(“InputMovie2.mov”);
inputs[3] = new VideoStill(“copyright.tif”, 5.0);
ExportMovie(inputs, “NewMovie.*”, settings);
```

Segment Method and Parameters

Use the `<movie>.setSegment(start, duration)` method to add a segment of the underlying movie to the output.

Syntax

```
var movie = new QuicktimeMovie(“Minnie.mov”);
movie.setSegment(2.5, 5.0);
```

Parameters

`start` - the start time, in seconds, of the segment.

`duration` - *(optional)* the duration, in seconds, of the segment. If the value is 0 or not set, the duration will be to the end of the movie.

Returns

Nothing

VideoStill Methods and Parameters

The `VideoStill` object is used to add a static frame of a specified duration to the output movie. There are three forms of the VideoStill constructor that return a VideoStill object.

new VideoStill(imagePath, duration)

Creates a `VideoStill` object representing the inclusion of a still image for a specified duration in the output movie.

Syntax

```
var still = new VideoStill("popeye.jpg", 2.0);
```

Parameters

`image` - The path of the image file to be included.

`duration` - the duration, in seconds, of the included image.

new VideoStill(media, duration)

Creates a `VideoStill` object representing the inclusion of a still image for a specified duration in the output movie.

Syntax

```
var still = new VideoStill(media, 2.0);
```

Parameters

`media` - Media object containing the image file to be included

`duration` - the duration, in seconds, of the included image

new VideoStill(width, height, color, duration)

Creates a `VideoStill` object representing the inclusion of a blank frame for a specified duration in the output movie.

Syntax

```
var blank = new VideoStill(128, 96, 0x00cc00, 1.5);
```

Parameters

`width` - The width of the frame.

`height` - The height of the frame.

`color` - The color, as an RGB integer value, of the frame.

`duration` - the duration, in seconds, of the frame.

ExportMovie: Output Path

The `outputPath` parameter specifies the path and name of the resulting movie file.

If the specified path ends with an asterisk ("*") character, the extension appropriate for the specified output container type is appended to the output path. Prior to appending the extension, the "*" is removed. If the path contains a period "." character, then any characters between the last "." and the end of the path will also be removed.

There are two ways to use this functionality:

- To add an extension without stripping off any existing extension, add ".*" to the end of the output path parameter.
- To replace an existing extension with the appropriate one for the output type, add just "*" to the end of the output path.

ExportMovie: Format and Compression Settings

The `settings` parameter provides settings that specify what kind of movie file will be created, including all the audio and video compression parameters. The parameter can be either a string specifying a file path to a file containing the QuickTime settings, an Array or Buffer object containing the settings, or an "on-the-fly" settings object that just specifies the output container type to use.

Before creating movie files using the `ExportMovie` function, one or more QuickTime or Windows Media (Windows only) settings files should be created or obtained and placed in a location that is accessible to the script calling the ExportMovie function. In most cases, the DeBabelizer application is used to create these files. In the case of Windows Media format settings for a Windows version of MediaRich, Microsoft's Windows Media Profile Editor tool is used instead. For information about creating these files, see "Using Compression Settings Files" on page 283.

A fairly complete set of settings files is included with MediaRich Audio/Video. If these files are sufficient for your needs, you can use them instead of having to create your own files.

It is possible to specify an export operation without having to have a settings file available. The "settings" parameter will accept an object that contains just the specification of the output container type to be used. This object should contain two member variables named "subType" and "manufacturer", the two bits of information QuickTime uses to specify a particular container. Each of these should be an integer value. In this case, the default compression settings for that container type are used. This is obviously very limiting. This functionality was added primarily for testing and for use while learning to use the system. It is not recommended that this method be used in a production environment.

The following is an example of using this "on-the-fly" output specification:

```
var settings = new Object();
settings.subType = numFrom4cc("MooV");
settings.manufacturer = numFrom4cc("appl");
```

This specifies the output of a QuickTime .mov file using the default compression settings, which for QuickTime 7 outputs a H264 compressed movie. The `numFrom4cc()` function is a utility function that converts a four-character string to an integer, to facilitate the use of the four-character codes routinely used in QuickTime programming.

ExportMovie: Process Object

The `processObject` parameter is optional and can be used to modify the export process in a number of ways. Use the `processObject` to specify the following:

- The width and height of the exported movie.
- A custom method for obtaining and modifying each video frame prior to export.

<processobject>.getOutputSize()

The `<processobject>.getOutputSize()` method allows the process object to specify the desired size of the output movie

Syntax

```
processObject.getOutputSize ();
```

Parameters

None.

Returns

An array containing two integer values specifying output width and height, or null.

If a process object is provided and it contains a `getOutputSize()` method, this method is called by ExportMovie to allow the process object to specify the desired size of the output movie. If the process object needs to specify the output size, this method should return an Array object containing two integers indicating the width and height (in that order) of the output movie. If this method returns null, ExportMovie uses other means to determine the output size (see “Determining the Output Movie Size” on page 283). If 0 is returned for either height or width, that value is computed from the other value to preserve the aspect ratio of the first input source.

Some QuickTime settings include a specification for width and height. In these cases, the specified video compressor will usually enforce that width and height by forceably scaling each frame to that size. Because of this, `getOutputSize()` methods will usually want to just return these values or null if they exist.

If the QuickTime settings for the export operation include a width and height, the “settingsOutputSize” member variable will be set on the process object prior to calling the `getOutputSize` method. This member variable will be an Array containing two integer values indicating the width and height (in that order) specified by the QuickTime settings.

The following is an example of using the `getOutputSize` method that allows for width and height values specified by the QuickTime settings. This method sets the output size to 640x480 if no width/height is specified in the QuickTime settings:

```
processObject.getOutputSize = function()
{
```

```
if (this.settingsOutputSize)
return null;

return [640, 480];
}
```

<processobject>.getFrame()

By supplying a process object with a `getFrame` method, the calling script can preprocess each input frame in any way it sees fit. It could, for example, composite a company logo or some copyright text onto each frame.

Syntax

```
<processobject>.getFrame(input, inTime, inFrame, outTime, outFrame);
```

Parameters

`input` - The current input object (a QuicktimeMovie or VideoStill object).

`inSegment` - The segment (both time and duration) of the input movie being requested.

`inFrame` - The current frame number in the input source.

`outTime` - the current time in the output movie.

`outFrame` - the current frame in the output movie.

Returns

A Media object.

Each time the `getFrame` method is called on the Process Object, it must return either media containing the next frame to be inserted into the movie, or NULL to indicate that no frame should be inserted at the current output time. If NULL is returned, the prior frame's duration is extended to make up for the missing frame.

The `inSegment` parameter is an object containing two fields; "`currentTime`" and "`duration`". Together, these fields specify the segment of the input movie that the `getFrame` method is expected to return. The "`duration`" field can be modified by the `getFrame` method to change the duration of the returned frame. This will cause the frame rate of the output movie to be changed.

The following examples help illustrate the use of the Process Object parameter.

Example 1

This example shows how to write a Process Object that simply returns the requested frame. Such an object has the same affect as if no Process Object was supplied.

```
var processObj = new Object();
processObj.getFrame = function(input, inTime, inFrame, outTime, outFrame)
{
return input.getFrameAtTime(inTime.currentTime);
}
ExportMovie(inputs, “result.mov”, settings, processObj);
```

Example 2

This example composites a logo at the top-left corner of each frame of the output video:

```
var processObj = new Object();
processObj.logo = new Media();
processObj.logo.load(name @ “ourLogo.tif”);
processObj.getFrame = function(input, inTime, inFrame, outTime, outFrame)
{
var frame = input.getFrameAtTime(inTime.currentTime);
frame.composite(source @ this.logo, handleX @ “left”,
handleY @ “top”);
return frame;
}
ExportMovie(inputs, “result.mov”, settings, processObj);
```

Keep in mind that the `getFrame` method of your Process Object is called many times during processing. You should make this function as efficient as possible. Notice that this example loads the logo file just once before calling `ExportMovie()`. If it loaded the logo file within the `getFrame` method, it would load the same file over and over again for every frame of the movie.

ExportMovie: Alternate Audio Track Inputs

The alternate audio track input array (`audioInputs` parameter) is optional. When it is not specified, the audio comes from the main inputs. This parameter can specify a string pathname to a file, a single `QuicktimeMovie` object input, or an array of `QuicktimeMovie` objects. This is the same as for the main inputs. The input CANNOT be a `VideoStill` or `VideoSource` object, because those do not contain audio.

If an array of inputs is specified, the inputs are appended to one another in the output movie, just as the main inputs are.

The inputs can be either an audio-video source or an audio source. If it is an audio-video source, the video track is ignored.

The main inputs dictate the overall time of the output movie. The alternate audio input is cropped to the length of the main inputs. If the alternate audio input is shorter than the main inputs then silence will be inserted until the main inputs are completely processed.

Example

```
ExportMovie("MyMainMovie.flv", "OutputMovie.mov",
"MySettingsFile.dat", null, "BGScore.mp3");
```

This creates an output movie and replaces the audio from MyMainMovie.flv with BGScore.mp3. The null parameter for `proc` indicates no custom frame callback is specified in this example.

Determining the Output Movie Size

`ExportMovie` uses the following progression to determine the size of the output movie:

1. If a process object was supplied and it contains a `getOutputSize` method, that method is called. If that method returns a non-null value, the return value is assumed to be an Array containing two integer values representing width and height, and those values are used as the width and height of the output movie.
2. If the QuickTime settings specify output width and height values, those values are used.

Using Compression Settings Files

Previously, the settings files in AVCore 1 consisted of QuickTime atom containers or Windows Media .prx files. For MediaRich 4.0 and AVCore 2, these are now all text-based JSON files and the parameters in the settings files are mostly standardized across handlers (this enables easy programmatic changes, or simple editing manually). In AVCore 1, most settings files accepted by the ExportMovie function can be created using the AVCore 2 settings generator on the MediaRich Server. If you have licensed MediaRich Audio/Video or are evaluating it, you should have this utility on your MediaRich server.

For information about creating these settings files using the AVCore 2 settings generator, see “Using AV Settings Files” on page 255.

After you have created one or more settings files, they must be made available to the MediaRich server. This is done in the same way any source asset is made available to MediaRich. For more information on where to place and how to access source asset files, see “File Systems” on page 39.

The Scene Sampler

Included with MediaRich Audio/Video Core 1.1 is a scene sampler module, named `SceneDetectFast.ms`. This module chooses a set of frames from a movie to try to provide a good representation of the content of the movie with just those frames. The module tries to avoid similar or “uninteresting” frames, such as blank areas at the beginning or end of the movie.

Scene Sampler Basic Usage

To make the `SceneDetectFast` object available for use in a MediaScript script, the file “sys:/SceneDetectFast.ms” must be included at the top of the script.

Basic use of the sampler involves two steps:

- A `SceneDetectFast` object is created. The movie to be operated on is passed into the `SceneDetectFast` constructor.
- The detect() method is called on that object. The number of frames that the detect() method should return times for is passed to the method. The method returns an Array() object that contains floating point time values, in seconds, of the times of the frames it has chosen.

The following is an example:

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

var inputMovie = new QuicktimeMovie("bozo.mov");
var detector = new SceneDetectFast(inputMovie);
var times = detector.detect(30);
```

This example chooses 30 representative frames from the "bozo.mov" movie. The "times" variable ends up pointing to an Array() object containing the times of those 30 frames.

Adjusting the Scene Sampler Behavior

The scene sampler behavior can be adjusted by passing a second parameter to the `detect()` method. This parameter should be a JavaScript object, as created with the "new Object()" construct. Various members of this object can be set before passing it to the detect() method to affect sampler behavior:

```
<object>.multiplier
```

This value determines how many frames are originally considered by the sampler. The sampler starts its operation by multiplying the requested number of frames by this multiplier to come up with the initial set of frames it will consider. It simply breaks the movie up into this number of chunks to select this number of evenly spaced frames to start with. The default value for this parameter is 10.

For example, if a value of 30 is passed to the detect() method as in the above example, the default behavior of the scene sampler will be to start with 300 evenly spaced frames from the movie. It will select the final 30 from this set of 300.

This parameter can be used to adjust the tradeoff between execution speed and precision. If the scene detector seems to be often missing parts of a movie that you think it should find, you can set this field to a number larger than 10 to correct this problem. However, the detection process will take longer after you do this.

```
<object>.darkCutoff
<object>.lightCutoff
```

The scene sampler throws out frames that it feels are close to completely black or completely white, assuming that these are uninteresting frames. These two values influence how the detector decides if a frame is "too light" or "too dark". For each frame, a "lightness" value is calculated that is between 0.0 and 1.0. Then the lightCutoff and darkCutoff values are used as thresholds to determine if the frame should be thrown away. If the frame's value is smaller than the value of darkCutoff or the frame's value is greater than the value of lightCutoff, then the frame is discarded. The default value for darkCutoff is 0.1 and the default value for lightCutoff is 0.9.

Adjusting these values will control which frames are discarded because they are too light or too dark. If you set darkCutoff to 0.0 and lightCutoff to 1.0 then no frames will be discarded because they are too light or too dark.

The following is the same as the previous example, but with all three of the values discussed in this section modified from their default values:

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

var inputMovie = new QuicktimeMovie("bozo.mov");
var detector = new SceneDetectFast(inputMovie);
var settings = new Object();
settings.multiplier = 20; // start with twice the frames
settings.darkCutoff = 0.2; // throw away more dark frames
settings.lightCutoff = 1.0; // dont' throw away light frames
var times = detector.detect(30, settings);
```

See "Example 9: Create an Animated GIF" on page 292 for an example of using the scene sampler to create an animated GIF file.

The AVCore 1.1 Examples

There are numerous AVCore 1.1 examples supplied as a series of files named "example1.ms" through "example9.ms". Also included are a sample input movie and a set of QuickTime Settings files that can be used with the examples and with your own scripts.

These examples are located in the MediaRichModules directory for your installation of MediaRich. The discussion that follows assumes that the reader has a basic knowledge of MediaRich and MediaScript programming. Specifically, it is assumed that the reader understands how the query parameters of an MRL map to script parameters in a MediaScript script. For information, see "HTTP API" on page 12 and "Post-Processing Parameters" on page 27.

Example structure

The examples read input files from and write output to a Virtual FileSystem (VFS) named "avikitex". This name is mapped to the contents of the MediaRich_AV directory that contains the example scripts. Scripts that write output to disk rather than returning it to the browser will write to a directory named Out within the MediaRich_AV directory. For information on setting up and using Virtual FileSystems, see "File Systems" on page 39.

The example URLs include the query parameter "nc=1". This parameter simply tells both MediaRich and the browser not to cache the results of the request. This makes it much more convenient to modify the example scripts and re-execute them to see the effects of the changes.

Many of the examples calculate a movie file that is stored to the disk and not returned to the browser. In these cases, it returns a small amount of HTML that displays "OK" in the browser at the end of the script, just to help the user know when the script completes execution. As this bit of code is not relevant to our discussion here, it will be left out of the example listings that follow to save space. #include and comment lines are left out for the same reason.

Running the examples

Each example includes comments at the top of its file describing that example. Included in those comments is a sample URL. If you open a web browser on the MediaRich Server where the samples are installed, you should be able to use this URL as is to execute each example.

If you want to run the examples from a browser running on a different machine, modify each example's URL by replacing "localhost" with the name or IP address of the MediaRich Server that is serving the examples.

Example 1: Simple Transcoder

This function does a simple transcoding of a single movie to a new file. Two string arguments are passed to the function; the path to the movie file to be processed, and the path to the QuickTime Settings file from which to obtain the compression settings to be used during the export.

Sample MRL

```
http://localhost/mgen/avikitex:/Scripts/example1.ms?args="iMac_
Dance.mov","Mov/V=H264,single"&nc=1
```

Script code

```
function main(input, settings)
{
ExportMovie("avikitex:/Media/" + input, "avikitex:/Out/example1/" + input + "*",
"avikitex:/QTSettings/" + settings);
}
```

In its simplest form, the `ExportMovie` function takes three parameters; a parameter defining the input movie or movies to be used during the export, the output file path indicating where to write the resulting movie, and the QuickTime Settings to use to do the export.

The movie file path passed to the `ExportMovie` function is assumed to be a path within the Media directory at the top level of the "avikitex" VFS. This is why the string "avikitex:/Media/" is prepended onto the path before passing it to the ExportMovie function. Similarly, the settings file path is assumed to be below the "avikitex:/QTSettings/" directory.

Output will be written to the "avikitex:/Out/example1/" directory. The name of the output movie will be the same as the input movie, except that the extension on the output file will be changed to match the kind of container being written to (as specified in the QuickTime Settings file). The addition of a "*" to the end of the output path tells ExportMovie to change the output file's extension to the proper one for the output format.

If the supplied sample MRL is used to run this example, the "iMac_Dance.mov" file will be used as the input movie, the "V=H264,single" file within the "Mov" directory will be read to get the QuickTime settings, and the resulting file will be written to "avikitex:/Out/example1/iMac_Dance.mov".

Example 2: Export a Portion of an Input Movie

This example is similar to Example 1. The only difference in behavior is that only a portion of the input movie is exported.

Sample MRL

```
http://localhost/mgen/avikitex:/Scripts/example2.ms?args="iMac_
Dance.mov","Mov/V=H264,single"&nc=1
```

Script code

```
function main(input, settings)
{
var inputObj = new QuicktimeMovie("avikitex:/Media/" + input);
inputObj.setSegment(5,4);

ExportMovie(inputObj, "avikitex:/Out/example2/" + input + "*",
"avikitex:/QTSettings/" + settings);
}
```

To accomplish this, a simple file path cannot be passed as the first parameter to `ExportMovie`. Instead, a `QuicktimeMovie` object is constructed from the input file path. Then, the `setSegment` method on that object is called to limit that object’s contribution to the output movie. In this case, only four seconds of the input movie is exported, starting at five seconds into the input movie. The `QuicktimeMovie` object is passed into the `ExportMovie` function in place of the input file path.

Example 3: Build a Movie from Multiple Sources

This example expands on the first two by exporting two segments of the input movie. It also inserts a blank frame between the two segments of the input movie that lasts for one second.

Sample MRL

```
http://localhost/mgen/avikitex:/Scripts/example3.ms?args="iMac_
Dance.mov","Mov/V=H264,single"&nc=1
```

Script code

```
function main(input, settings)
{
var inputs = new Array();

inputs[0] = new QuicktimeMovie("avikitex:/Media/" + input);
inputs[0].setSegment(0,4);

inputs[1] = new VideoStill(inputs[0].getWidth(), inputs[0].getHeight(), 0x0000ff,
1.0);
inputs[2] = new QuicktimeMovie("avikitex:/Media/" + input);
inputs[2].setSegment(8,4);

ExportMovie(inputs, "avikitex:/Out/example3/" + input + "*", "avikitex:/QTSettings/"
+ settings);
}
```

The key concept here is that `ExportMovie` can take an Array object as the first parameter, where each object in that array is either a `QuicktimeMovie` object or a `VideoStill` object. The objects in the array are taken in order to produce the resulting movie.

So here, it first creates an empty Array object. It then creates a `QuicktimeMovie` object that refers to the input movie and puts that object into the first slot in the Array. It calls the `setSegment` method on that object to so that the first four seconds of the input movie is written to the output.

Next, it creates a `VideoStill` object and puts it in the second slot in the Array. In this case, it's creating a still frame with the same dimensions as the input movie. The color of the frame is dark blue (the color specification is of the form 0xRRGGBB), and the frame remains on the screen for one second when the movie is played at normal speed.

It then adds a third object to the Array, this time specifying a different segment of the input movie.

And finally, it calls `ExportMovie` as before, but passing the Array as the first parameter.

Note: This function would work equally well if the second QuickTime movie object were constructed from a different input movie file. The input Array can contain input segments from any number of movie files.

Example 4: Change the Output Movie Dimensions

This example is similar to Example 1, except that the output movie is scaled to a particular frame size. To specify the size of the output movie, an object is created and passed as the fourth parameter in the call to ExportMovie. This fourth parameter, when specified, is always an Object. Depending on what members exist in this Object, the export operation is modified in various ways.

Sample MRL

```
http://localhost/mgen/avikitex:/Scripts/example4.ms?args="iMac_
Dance.mov","Mov/V=H264,single"&nc=1
```

Script code

```
function main(input, settings)
{
var processObject = new Object();
processObject.getOutputSize = function()
{
if (this.settingsOutputSize[0] > 0)
return null;
return [ 50, null ];
}
ExportMovie("avikitex:/Media/" + input, "avikitex:/Out/example4/" + input + "*",
"avikitex:
{
QTSettings/" + settings, processObject);
}
```

To modify the size of the output movie, it creates an object and adds a `getOutputSize` method to that object. That method should return an Array with two values in it to indicate the width and height, respectively, of the output movie. If "`null`" is passed for one of those two values, that value is computed to preserve the aspect ratio of the input movie based on the other value.

In some cases, an output width and height are specified in the QuickTime Settings file. When this happens, it does no good to specify a different size for the output movie. No matter what the system does, QuickTime will ultimately resize the movie to match the width and height that were specified in the settings. This is why the sample code checks the value of "`this.settingsOutputSize`".

Before `getOutputSize` is called, the system sets the value of the "`settingsOutputSize`" member variable of the passed-in object to indicate if a width and height were specified in the QuickTime Settings file. If they are, this variable contains an Array containing two integers that indicate the width and height specified in the settings. If no width and height are specified in the settings, this Array will contain -1 for both the width and the height.

The sample `getOutputSize` method checks to see if a valid width value exists in the `settingsOutputSize` variable. If so, the method returns "null" to indicate that it does not specify an output size. Otherwise, the method returns a size specification.

The sample code returns a value of "[50, null]", which indicates that the output movie width should be 50 pixels, and the height should be whatever it needs to be to preserve the aspect ratio of the input movie. For example, if the input movie had dimensions 200x100, the output movie has dimensions 50x25.

Example 5: Process Video Frames (Draw)

This example is similar to Example 1. The difference is that draws the current frame number on each of the frames of the movie. This is accomplished by creating an Object, adding a `getFrame` method to it, and passing that object as the fourth parameter to the ExportMovie function.

Sample MRL

```
http://localhost/mgen/avikitex:/Scripts/example5.ms?args="iMac_
Dance.mov","Mov/V=H264,single"&nc=1
```

Script code

```
function main(input, settings)
{
var processObject = new Object();
processObject.getFrame = function(inMovie, inTime, inFrame, outTime, outFrame)
{
var frame = inMovie.getFrameAtTime(inTime.currentTime);
frame.drawText({text:outFrame, size:64, x:5, y:5, handlex:"left", handley:"top",
smooth:true});
return frame;
}
ExportMovie(inputObj, "avikitex:/Out/example5/" + input + "*",
"avikitex:/QTSettings/" + settings, processObject);
}
```

A `getFrame` method takes five parameters. The first parameter is the current input movie being read. The second and third parameters are the current time and frame number, respectively, in the input movie. The fourth and fifth parameters are the current time and frame number, respectively, in the output movie.

In this case, it reads the requested frame from the input movie by calling the `getFrameAtTime` method on the input movie. This is a best practice in almost all cases. The `getFrameAtTime` method returns a Media object.

After it has the Media object, it can apply any operation supported by the Media object. In this case, it uses the Media object `drawText` method to draw the frame number onto the frame.

The `getFrame` method is expected to return a Media object, so it simply returns the Media object it just read and then modified.

Example 6: Process Video Frames (Rotate)

This example is very similar to Example 5. The only difference is that instead of drawing onto each frame, it rotates each frame based on the current output frame number. This causes the input movie to spin around in the new movie. This demonstrates that you can do just about anything to the output frame.

Sample MRL

```
http://localhost/mgen/avikitex:/Scripts/example6.ms?args="iMac_
Dance.mov","Mov/V=H264,single"&nc=1
```

Script code

```
function main(input, settings)
{
var inputObj = new QuicktimeMovie("avikitex:/Media/" + input);
var processObject = new Object();
processObject.getFrame = function(inMovie, inTime, inFrame, outTime, outFrame)
{
var frame = inMovie.getFrameAtTime(inTime.currentTime);
frame.rotate({angle:outFrame,smooth:true});
return frame;
}
ExportMovie(inputObj, "avikitex:/Out/example6/" + input + "*",
"avikitex:/QTSettings/" + settings, processObject);
}
```

Example 7: Grab Single Frames from a Movie

This example is unlike any of the others. Instead of creating a new movie file, this function causes a single frame of the input movie to be returned to the browser. The function accepts a movie name as its first parameter, but the second specifies the place in the input movie, in seconds, from which to grab the frame.

Sample MRL

```
http://localhost/mgen/avikitex:/Scripts/example7.ms?args="iMac_Dance.mov",1.5&nc=1
```

Script code

```
function main(input, time)
{
var inputObj = new QuicktimeMovie("avikitex:/Media/" + input);
var frame = inputObj.getFrameAtTime(time);
frame.save({type:"JPEG"});
}
```

It starts by creating a QuicktimeMovie object and passing it the path to the input movie. Then it calls the `getFrameAtTime` method on this object, passing it the time parameter. This method returns a Media object.

To cause the Media object to be returned to the client browser, it calls the object `save()` method. The parameter to the save method is the type of image we want returned.

The sample MRL above will cause the frame at 1.5 seconds in the iMac_Dance.mov movie to be returned to the browser. By changing the number at the end of the MRL, different frames of that same movie can be returned and viewed.

Example 8: Extract Movie Information

This example expands on Example 7. Instead of returning a frame of the input movie, this function returns a composite image containing a thumbnail from the movie and some additional information about the movie—the movie width and height, duration, frame count, and frame rate.

Sample MRL

```
http://localhost/mgen/avikitex:/Scripts/example8.ms?args="iMac_Dance.mov",1.5&nc=1
```

Script code

```
function main(input, time)
{
var inputObj = new QuicktimeMovie("avikitex:/Media/" + input);
var frame = inputObj.getFrameAtTime(time);
frame.scale({xs:120,constrain:true});
var result = new Media();
result.makeCanvas({xs:400, ys:10 + frame.getHeight()});
result.composite({source:frame, x:5, y:5, handlex:"left", handley:"top"});
var xpos = 130;
var ypos = 5;
var text = "Width: " + inputObj.getWidth() + " Height: " + inputObj.getHeight();
result.drawText({x:xpos, y:ypos, handlex:"left", handley:"top", size:22, text:text});
ypos = 30;
text = "Duration: " + inputObj.getDuration().toFixed(2);
result.drawText({x:xpos, y:ypos, handlex:"left", handley:"top", size:22, text:text});
ypos = 55;
text = "Frames: " + inputObj.getFrameCount() + " FPS: " + (inputObj.getFrameCount() /
inputObj.getDuration()).toFixed(2);
result.drawText({x:xpos, y:ypos, handlex:"left", handley:"top", size:22, text:text,
smooth:true});
result.save({type:"JPEG"});
}
```

First, it extracts a frame from the movie and scales it to the desired thumbnail size. Then, it creates a blank image that is a little taller than the thumbnail and has extra width, so that there is an area for drawing some text.

Next, it composites the thumbnail image onto the blank image. Most of the remaining code involves extracting information from the input movie and drawing text using that information onto the output frame. This information is obtained by calling various methods on the QuicktimeMovie object representing the input movie.

Finally, it calls the save() method of the output Media to send the final frame back to the caller.

Example 9: Create an Animated GIF

This example extracts a series of frames from an input movie to produce an animated GIF that represents the movie. It also demonstrates using the SceneDetectFast module to choose the frames to be extracted.

The inputs to this function are the input movie, the number of frames that the GIF will contain, and the amount of time to delay between each frame in the GIF. If the delay parameter is omitted or is 0, then the output GIF’s pace will be matched to that of the input movie.

Sample MRL

```
http://localhost/mgen/avikitex:/Scripts/example9.ms?args="iMac_Dance.mov",30,.25&nc=1
```

Script code

```
function main(input, count, delay)
{
var inputObj = new QuicktimeMovie("avikitex:/Media/" + input);
var detector = new SceneDetectFast(inputObj);
var times = detector.detect(count);
var gif = new Media();
for (i = 0 ; i < count ; i++)
{
var frame = inputObj.getFrameAtTime(times[i]);
gif.frameAdd({source:frame});
}
gif.scale({xs:96, constrain:true});
gif.reduce();
if (!delay)
delay = inputObj.getDuration() / (count-1);
gif.save({type:"gif", delay:(delay*100)});
}
```

First, it creates a QuicktimeMovie object from the input movie. Then it computes the time delay between frames to break the movie into equal-size chunks based on the count parameter.

It creates a new Media object to contain the output GIF. Then the function loops, extracting each of the desired frames and adding them to the GIF object.

It then scales the GIF to the desired size, and calls reduce() to reduce each frame of the GIF to an 8-bit image (a requirement of the GIF format).

Finally, it saves the GIF to send it back to the caller. In this case, it passes an extra parameter to the `save()` method to specify the delay between frames of the GIF when it is displayed.

CHAPTER 7

MediaRich Proof Sheet Generator

The Proof Sheet Generator is a MediaRich extension module. It adds an `_MR_ProofSheet` object to the MediaScript language. This object exposes a the `generate` method, which takes an XML file as input and produces a proof sheet PDF document.

Chapter summary

The generate Method

The following is an example of calling the Proof Sheet Generator within a MediaRich script:

```
#link "ProofSheet.dll"
var ps = new _MR_ProofSheet();
var result = ps.generate("input.xml", "result.pdf”, "errors.txt");
```

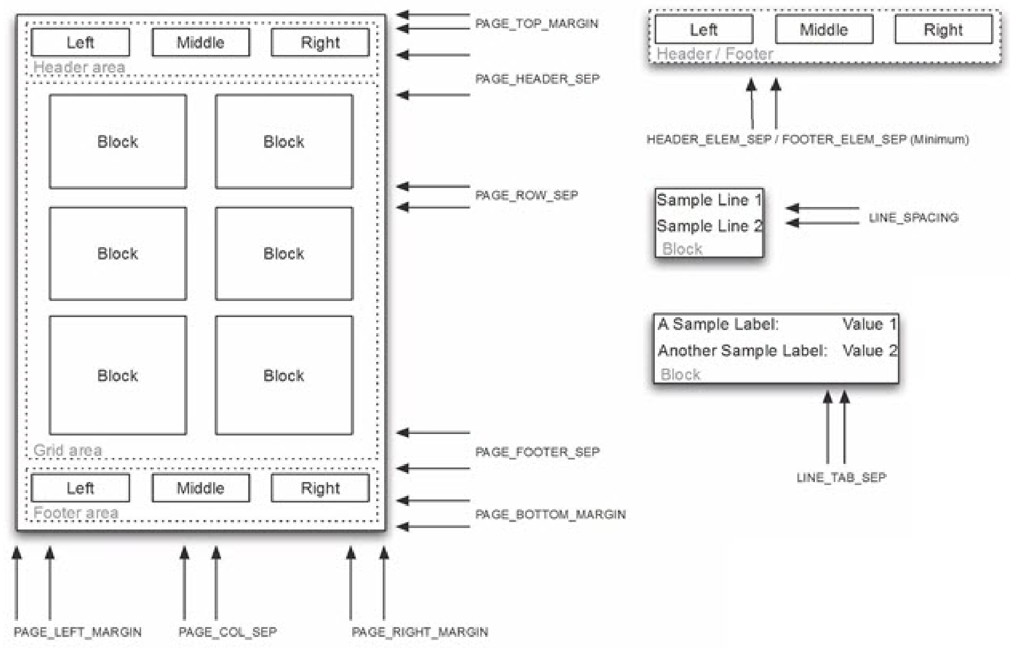
In this standard usage of the `generate()` method, the first parameter specifies the file that contains the XML specification of the proof sheet to be generated. The second parameter specifies a file name for writing the resulting PDF file. The third parameter, which is optional, specifies a file name to use for writing any error messages or other diagnostic information generated during the creation of the proof sheet. If the third parameter is omitted, no diagnostic information is returned.

Proof Sheet Layout

A proof sheet consists of one or more pages. Each page consists of three areas: the Header area, the Body area, and the Footer area.

The Header and Footer areas each consist of three blocks: the Left block, the Middle block, and the Right block. The Body area consists of rows and columns of blocks between the Header and Footer.

The following diagram provides a visual representation of the layout of a Proof Sheet, including the Header, Body, and Footer areas. This diagram also illustrates many of the spacing parameters used to precisely define the layout of a Proof Sheet. You should refer to this diagram as you review the remainder of this document.



Defining the Proof Sheet XML Input

A Proof Sheet Specification is an XML document that provides all the information necessary to build a Proof Sheet. This information can be broken down into the following categories.

Structural Information

The structure of the XML that allows the other parts of the XML document to apply to specific areas of the proof sheet.

Parameters

Each parameter is a named value (key/value pair) that provides the parameter value used to construct some portion of the proof sheet.

Text Content

This is the literal text to appear on each page of the proof sheet

Image References

These are file system paths that reference image files containing the images that appear in the proof sheet.

Proof Sheet Structure

The structure of the XML document is a hierarchy of container objects. The outer container is the entire proof sheet. The next level consists of the Header, the Body area, and the Footer. Each of these areas contain Blocks. Each block consists of one or more Text Lines and/or Image Lines.

Any level of the XML hierarchy can contain a parameter block containing one or more parameters. The parameters in a parameter block apply to the parent block that contains that parameter block. Those parameters apply to all the child blocks contained in the parent block as well. If the same parameter appears at multiple levels of the container hierarchy, parameters at the deeper level override parameters at the higher level. This allows you to specify default parameters that apply to all areas of a proof sheet, but then override those defaults for specific areas on the sheet.

The following XML tag outline shows all possible block and param tag levels:

```
<PROOFSHEET>
    <PARAMS></PARAMS>
    <HEADER>
        <PARAMS></PARAMS>
        <LEFT>
            <PARAMS></PARAMS>
            <TEXTLINE>
                <PARAMS></PARAMS>
            </TEXTLINE>
        </LEFT>
        <CENTER>
            <PARAMS></PARAMS>
            <TEXTLINE>
                <PARAMS></PARAMS>
            </TEXTLINE>
        </CENTER>
        <RIGHT>
            <PARAMS></PARAMS>
            <TEXTLINE>
                <PARAMS></PARAMS>
            </TEXTLINE>
        </RIGHT>
    </HEADER>
    <BLOCKS>
        <PARAMS></PARAMS>
        <BLOCK>
            <PARAMS></PARAMS>
            <TEXTLINE> or <IMAGELINE>
                <PARAMS></PARAMS>
            </TEXTLINE> or </IMAGELINE>
        </BLOCK>
    </BLOCKS>
    <FOOTER>
        <PARAMS></PARAMS>
        <LEFT>
            <PARAMS></PARAMS>
            <TEXTLINE>
                < PARAMS></PARAMS>
            </TEXTLINE>
        </LEFT>
        <MIDDLE>
            <PARAMS></PARAMS>
            <TEXTLINE>
                <PARAMS></PARAMS>
            </TEXTLINE>
        </MIDDLE>
        <RIGHT>
            <PARAMS></PARAMS>
            <TEXTLINE>
                <PARAMS></PARAMS>
            </TEXTLINE>
        </RIGHT>
    </FOOTER>
</PROOFSHEET>
```

The information specified within the enclosing tags for each area (`<HEADER> </HEADER>`, `<BLOCKS> </BLOCKS>`, or `<FOOTER> </FOOTER>`) determines the content of that area. Within these enclosing tags are enclosing sub-tags for each of the blocks that can be displayed in that area. If you do not include a tag for a particular area or block, the Proof Sheet Generator does not render that element.

Only the `<PROOFSHEET> </PROOFSHEET>` enclosing tags are required. All other tags are optional. However, with only these two tags, all you are going to generate is a blank page.

Proof Sheet Page Layout Modes

The `PAGE_LAYOUT` parameter specifies how space on the page is allocated to the blocks in the body section of a Proof Sheet, using the following layout modes:

Mode tag	Description
`Standard`	Allows for blocks to be enlarged when there is available page space not used by neighboring blocks This can result in images and text on the page to be larger than they would otherwise be, thereby making them more readable. On the last page, where extra space can occur at the bottom of the page due to fewer blocks appearing on this page, the extra space is *not* used by the blocks on the page. This is the default layout mode if one is not specified explicitly.
`StretchLastPage`	Produces the same results as the standard layout mode, except that any extra space on the last page is used by other blocks on the page to allow those blocks to be larger
`Fixed`	Ensures that blocks are never enlarged to utilize extra space on the page Each block on the page is allotted the same amount of space, and no block more than this amount. This mode produces the most uniform looking Proof Sheet, but tends to result in sheets with more blank area on the page.

If the contents of each block in a Proof Sheet is the same size, the `Standard` and `Fixed` layout modes produce the same results. In this case, the only difference is how the last page is laid out.

Proof Sheet Page Spacing

Specific parameters are used to indicate the layout of the page and the spacing that occurs between the various blocks on a page. Each of these parameters, except for `PAGE_ROWS` and `PAGE_COLUMNS`, is specified in inches. The following table describes these parameters:

Parameter tag	Description
`PAGE_WIDTH`	Specifies the total width of the page, including margins
`PAGE_HEIGHT`	Specifies the total height of the page, including margins
`PAGE_ROWS`	Specifies the total number of rows of Grid blocks on a page
`PAGE_COLUMNS`	Specifies the total number of columns of Grid blocks on a page
`PAGE_LEFT_MARGIN`	Specifies the amount of space between the left edge of page and page blocks
`PAGE_TOP_MARGIN`	Specifies the amount of space between the top edge of page and page blocks
`PAGE_RIGHT_MARGIN`	Specifies the amount of space between the right edge of page and page blocks
`PAGE_BOTTOM_MARGIN`	Specifies the amount of space between the bottom edge of page and page blocks
`PAGE_HEADER_SEP`	Specifies the amount of space between the header blocks and grid content blocks
`PAGE_FOOTER_SEP`	Specifies the amount of space between the grid content blocks and footer blocks
`PAGE_ROW_SEP`	Specifies the minimum amount of space between each row of blocks in the grid content area
`PAGE_COL_SEP`	Specifies the minimum amount of space between each column of grid content blocks in the grid area
`HEADER_ELEM_SEP`	Specifies the minimum amount of space between each header block
`FOOTER_ELEM_SEP`	Specifies the minimum amount of space between each footer block

Proof Sheet Text Lines

The blocks within each of the proof sheet areas can contain any number of lines. Each line consists of either text or an image.

The `<TEXTLINE>` enclosing tags are used to specify a line of text. The following parameters can be applied to the line of text to control the look and placement (justification) of the text:

Parameter tag	Description
`FONT_FAMILY`	Specifies the font family name (string)
`FONT_STYLE`	Specifies the font style (string) Valid values are NORMAL, BOLD, ITALIC, or BOLDITALIC. If not specified, the default (NORMAL) is used.
`FONT_SIZE`	Specifies the font size, in points (float)
`FONT_MINSIZE`	Specifies the minimum font size, in points (float) Text lines, like image lines, are scaled to fit cleanly on the page. In some cases, you might prefer to allow text to flow into the margins and neighboring blocks rather than scaling to a size that is unreadable. When specified, this parameter sets the minimum post-scaling size of the text and the text to which it is applied is never smaller than this value.
`LINE_JUSTIFY`	Specifies the line justification (string) Valid values are LEFT, CENTER, or RIGHT.
`LINE_TAB_SEP`	Specifies the line tab separation, in inches (float)

The `FONT_FAMILY`, `FONT_STYLE`, and `FONT_SIZE` parameters specify the look and size of the text. The `LINE_JUSTIFY` parameter defines how the text is justified within a block. By default, text is left justified.

When one or more text lines in a block are left justified and contain a tab character, those text lines are split at the tab character to define two columns of text. The portion of each line that appears after the tab character is left justified into a second column. The `LINE_TAB_SEP` parameter defines the amount of space between the two columns of text (the amount of space between the end of the widest line in the first column and the beginning of the second column). This parameter, like all spacing parameters, is specified in inches.

Proof Sheet Image Lines

The `<IMAGELINE>` enclosing tag is used to specify an image line within a block. To be useful, this tag always contains exactly one `<FILE>` tag which points to the image to be included at that point in the Proof Sheet.

The following parameters can be applied to an image line to effect its size and placement:

Parameter tag	Description
`IMAGE_DPI`	Specifies the minimum image DPI for the image (float) The image is first reduced in size as necessary to fit it properly within its area of the page. If, after this size reduction, the image's resolution is smaller than the specified minimum DPI, the image's size is further reduced to force it to this minimum DPI. This parameter is primarily used when the resulting PDF is going to be printed, to insure that the image has enough resolution to look good when printed. **Note:** Images are never scaled up in size. That is, they never appear at a lower resolution than their native resolution (usually 72 DPI).
`LINE_JUSTIFY`	Specifies the line justification (string) Valid values are LEFT, CENTER, or RIGHT.

Vertical spacing between lines

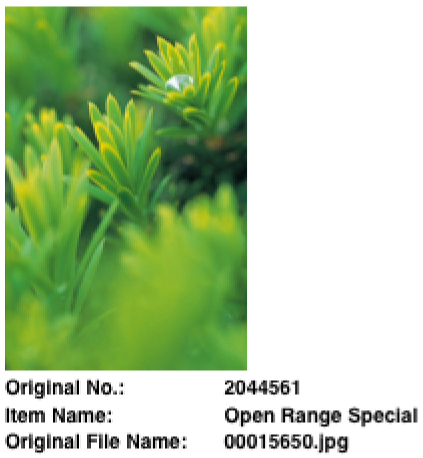
The `LINE_SPACING` parameter is used to define the amount of space between each line in a block. This parameter is interpreted in inches.

Proof Sheet Input Examples

The following example specifies the content for a single Grid Block consisting of an image followed by three lines of text:

```
<BLOCK>
    <PARAMS>
        <LINE_JUSTIFY>Left</LINE_JUSTIFY>
    </PARAMS>
    <IMAGELINE>
        <FILE>c:\Library\Images\comstock/00015650.jpg</FILE>
    </IMAGELINE>
    <TEXTLINE>
        <TEXT>Original No.: 2044561</TEXT>
    </TEXTLINE>
    <TEXTLINE>
        <TEXT>Item Name: Open Range Special</TEXT>
    </TEXTLINE>
    <TEXTLINE>
        <TEXT>Original File Name: 00015650.jpg</TEXT>
    </TEXTLINE>
</BLOCK>
```

The example renders the following block within the Grid Blocks area of the page:

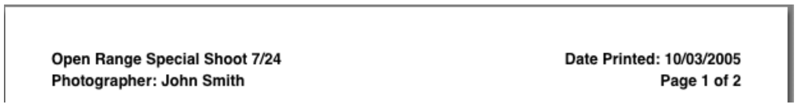


Original No.:	2044561
Item Name:	Open Range Special
Original File Name:	00015650.jpg

The following example specifies the content for a Header area. It defines the Left and Right blocks, but does not specify the Center block. It specifies a single font face and size that applies to the entire Header area:

```
<HEADER>
    <PARAMS>
        <FONT_FAMILY>Arial</FONT_FAMILY>
        <FONT_SIZE>12</FONT_SIZE>
    </PARAMS>
    <LEFT>
        <TEXTLINE>
            <TEXT>Open Range Special Shoot 7/24</TEXT>
        </TEXTLINE>
        <TEXTLINE>
            <TEXT>Photographer: John Smith</TEXT>
        </TEXTLINE>
    </LEFT>
    <RIGHT>
        <PARAMS>
            <LINE_JUSTIFY>RIGHT</LINE_JUSTIFY>
        </PARAMS>
        <TEXTLINE>
            <TEXT>Date Printed: {TODAY}</TEXT>
        </TEXTLINE>
        <TEXTLINE>
            <TEXT>Page {PAGE} of {PAGES}</TEXT>
        </TEXTLINE>
    </RIGHT>
</HEADER>
```

This renders the following header at the top of the page:



Open Range Special Shoot 7/24
Photographer: John Smith

Date Printed: 10/03/2005
Page 1 of 2

The following example illustrates a Body section that displays four images, with a line of text below each image indicating the image's file name. A single `LINE_JUSTIFY` parameter is specified so that each image appears in the center of its area of the page, and each text label appears centered below its image.

```
<BLOCKS>
    <PARAMS>
        <LINE_JUSTIFY>Center</LINE_JUSTIFY>
    </PARAMS>
    <BLOCK>
        <IMAGELINE>
            <FILE>C:\MyImages\Flower.jpg</FILE>
        </IMAGELINE>
        <TEXTLINE>Flower.jpg</TEXTLINE>
    </BLOCK>
    <BLOCK>
        <IMAGELINE>
            <FILE>C:\MyImages\Dog.jpg</FILE>
        </IMAGELINE>
        <TEXTLINE>Dog.jpg</TEXTLINE>
    </BLOCK>
    <BLOCK>
        <IMAGELINE>
            <FILE>C:\MyImages\Chair.jpg</FILE>
        </IMAGELINE>
        <TEXTLINE>Chair.jpg</TEXTLINE>
    </BLOCK>
    <BLOCK>
        <IMAGELINE>
            <FILE>C:\MyImages\Clown.jpg</FILE>
        </IMAGELINE>
        <TEXTLINE>Clown.jpg</TEXTLINE>
    </BLOCK>
</BLOCKS>
```

Generating Custom Images for the Proof Sheet

As an alternative to using existing images for the proof sheet grid blocks, you can generate the images so that they are sized specifically for the proof sheet. This provides the ability to create a proof sheet for a set of images using a specific DPI for all of the images in the PDF document.

You can do this by creating a MediaRich script that includes a special usage of the `generate()` method. When the `generate()` method is passed a file name in its second parameter that ends in ".xml", it does not produce a PDF file. Instead, it computes a size for each image in the Proof Sheet. If images are provided at these sizes when the Proof Sheet is rendered, all images in the sheet are a consistent DPI. This size information is returned as a modified version of the passed in Proof Sheet Specification XML document, saved to disk at the path passed in the second parameter to the method. This new document is a copy of the original document, with image size tags added to each of the `<IMAGELINE>` tags.

The following is an example of the XML file generated by the `generate()` method:

```
<?xml version="1.0" encoding="UTF-8"?>
<PROOFSHEET>
    <BLOCKS>
        <BLOCK>
            <IMAGELINE>
                <WIDTH>121</WIDTH>
                <HEIGHT>181</HEIGHT>
                <FILE>c:\Library\Images\comstock/
                015650.jpg</FILE>
            </IMAGELINE>
        </BLOCK>
……
    </BLOCKS>
</PROOFSHEET>
```

To use this feature, a MediaScript script first calls `generate()` to compute the image sizes and return a new XML document containing them. The script then reads the XML file to extract the proper size for each image, and pre-processes the source images to the sizes indicated by that document. Finally, it calls `generate()` a second time to actually render a PDF document, this time passing the modified XML document as the first parameter.

CHAPTER 8

MediaRich Hot Folders

The MediaRich Hot Folders mechanism allows batch processing jobs to be initiated by copying source media files into a specific directory that MediaRich is monitoring. When files are dropped into a Hot Folder, MediaRich processes the files according to the one or more scripts associated with that Hot Folder. After processing, the original files are either moved to a "processed" directory or deleted.

The Hot Folder mechanism supports some conventions as to where the results of processing are placed and how errors are reported, but the Hot Folder script is free to write results and status information to any location accessible by MediaRich.

Chapter summary

Working with Hot Folders

MediaRich looks for Hot Folders below preconfigured Hot Folder Root directories. Any number of Hot Folders can be created below any of these Hot Folder Root directories. These Hot Folder areas can exist on any filesystem accessible to MediaRich. They will often be placed on network accessible filesystems so that batch processing can be initiated from workstations on the network by mounting that same filesystem and dropping media files into the Hot Folders defined there.

The Hot Folder mechanism provides the following services to the MediaScript scripts that determine what a specified Hot Folder will do when files are dragged into it:

- Ability to define "pre" and "post" operations that are run prior to and after a batch of files is processed.
- Access to the location of each file to be processed
- Access to standard output locations
- Standard mechanism for reporting errors and for writing out diagnostic information
- Ability to send email messages to report errors and/or job success, and to attach results to such email messages as enclosures.
- Mechanism for retaining original files or for deleting them after the job completes

Defining Hot Folder Root Directories

Define Hot Folder Root directories using the MediaRich CORE Administration tool. When you open this tool and log into the server to be configured, stop the server if necessary, and then click the **Configure...** button to display the configuration dialog and select the **Hot Folder Roots** tab.

This tab displays currently defined Hot Folder Roots and allows for adding new roots and editing existing ones. After you use the tool to modify the Hot Folder Roots configuration, start the server.

Defining a Hot Folder

After the Hot Folder root directories are configured, you can define any number of hot folders. Generally, you create a separate Hot folder for each type of automated processing that you to put into place.

To define a Hot Folder:

1. Create a directory somewhere below an existing Hot Folder Root directory.
2. Create one or more MediaScript script files in that directory, named appropriately.

> ***Warning:*** As soon as these first two steps are complete, MediaRich will start processing files with this Hot Folder. You must ensure that there are not any other "processable" files in this folder when you create the first script file or MediaRich will immediately attempt to process those files.

3. Edit each script file to define the operation(s) that the Hot Folder will perform.

The following sections provide detailed information about creating these script files.

Naming MediaScript Script Files

Two conventions must be followed when naming the MediaRich scripts that define a Hot Folder operation. First, the script name must end with ".ms", which is the standard MediaScript file extension. Second, the first character of the name must be "^" to designate it as a non-processable file.

The following are examples of correct MediaScript script file names:

- ^Scale320x240.ms
- ^script.ms
- ^blahblah.ms

The following are examples of incorrect MediaScript script file names:

- Rotate90.ms (does not start with "^")
- ^yoyo.txt (does not end with ".ms")

Processable Hot Folder Files

A *processable* is any file that is not specifically ignored by the Hot Folder mechanism. The Hot Folder mechanism ignores any file or directory at the top level of a Hot Folder with a name that starts with a "^" (caret) character. This naming scheme is used by the Hot Folder mechanism itself to place control files and directories within the Hot Folder that it does not want processed. This is why the script files for the Hot Folder must have names that start with "^". This makes it clear that the script file itself is not processed as an input file. The Hot Folder user can make use of this naming feature to place other files in the Hot Folder that will be ignored by MediaRich.

Given the Hot Folder at the path "C:/Hot Folders/My Hot Folder", the following are some examples of processable files:

- C:/Hot Folders/My Hot Folder/horse.gif
- C:/Hot Folders/My Hot Folders/animals/pig.jpg
- C:/Hot Folders/My Hot Folders/animals/birds/parrot.tif

The following are examples of non-processable files:

- C:/Hot Folders/My Hot Folder/^script.ms
- C:/Hot Folders/My Hot Folder/^dont_process_these/whale.jpg
- C:/Hot Folders/My Hot Folder/^hiding/animals/dog.tga

As these examples imply, any level of file and directory nesting can exist in the files that are dragged into a Hot Folder. The Hot Folder mechanism makes it easy for the Hot Folder script to preserve these nested relationships in the output that it generates, but the script could also flatten the hierarchy in the output if it is designed to do so.

The Hot Folder Processing Sequence

When a Hot Folder is defined, Hot Folder processing is "triggered" whenever one or more "processable" files are copied into it and no additional changes to the contents of the Hot Folder occur within about ten seconds.

When a Hot Folder triggers, the following occurs for each script in the hot folder:

1. A batch ID is assigned to this batch of files.

   To determine the batch ID, MediaRich looks for a file named "^nextBatchId" in the Hot Folder. If this file exists, MediaRich reads it to determine the next batch ID. If no such file exists, a batch ID of "1" is used. In either case, the next batch ID to use is written into the "^nextBatchId" file. This means that the next batch ID to be used for a Hot Folder job can be specified by deleting the ^nextBatchId file and its corresponding numbered folders in the ^Processed and ^Results folders.
2. The "processable" files, including any directory hierarchy in which they are contained, are moved to the directory "^Processed/<batch ID>" directory within the Hot Folder, where <batch ID> is replaced with the batch ID determined in step 1.
3. The "hf_pre" function is called in the Hot Folder script file.
4. The "hf_file" function is called repeatedly, once for each "processable" file.
5. The "hf_post" function is called.
6. If the script file has specified that email should be sent either to report errors or to provide successful job completion notification and/or results, the email is sent.
7. If the "hf_post" function requests that the original files be deleted, those files are deleted.

Error reporting

If errors occur during processing, they are reported in a text file named "<batch ID>_Errors" that is placed in the "^Results" directory, where <batch ID> is replaced with the batch's ID.

Multiple Script Files in a Single Hot Folder

There are two reasons for placing multiple script files in a single Hot Folder:

- Convenience – If you have a collection of simple scripts that perform single actions, you can easily build Hot Folders that perform a combination of these single actions by copying the script file for each action into the Hot Folder.
- Parallelism – A single invocation of a script will always run on a single processor. If it performs multiple actions, those actions occur in a serial fashion and do not make use of spare processor resources. By breaking up a single script into multiple smaller scripts, each of those scripts can run simultaneously, possibly making better use of available processor resources.

If multiple script files exist in a Hot Folder, step 2 of the processing sequence is only performed for the first script. Additional scripts refer to the source files in the location determined by the Batch ID of the first script.

It is not possible to delete the original files when processing multiple script files. This is due to the fact that one of the script files must be responsible for deleting the originals, but that script has no way of waiting to be sure that the other scripts have run to completion before doing the cleanup. If this is a problem, you can use a second mechanism called "Phasing" to accommodate parallelism for a script while avoiding this issue. For information on Phasing, refer to the section called "Processing in Multiple Phases" later in this document.

Standard Hot Folder Output Locations

There are two standard output locations that are computed and made available for use by a Hot Folder script. The first location is called "Shared" and the second location is called "PerBatch". A script can use either of these locations for output, or save output to another location according to your custom requirements.

Shared

The "Shared" output location is common to all batches of files processed by the Hot Folder. That is, if a Hot Folder writes results to the Shared Location, one batch job can overwrite the results of a prior batch job if it writes files with the same names. The "Shared" output location is the directory named "^Results" at the top level of the Hot Folder.

PerBatch

The "PerBatch" output location is a subdirectory named with the batch ID within the "^Results" directory; for example, if the batch ID is 24, the "PerBatch" output location is "^Results/24". Using the "PerBatch" output location ensures that the results for each batch job are kept separate.

Hot Folder Script Structure

A Hot Folder Script is what determines the operations that a Hot Folder will perform on the files dragged into it. A Hot Folder script always has a file name of the form "^XXX.ms", where XXX can be replaced by any valid file name characters. The basic structure of a Hot Folder script file looks similar to the following:

```
// HF_PARAM1 = Param Value #1
// HF_PARAM2 = Param Value #2
// HF_PARAM3 = Param Value #3

function hf_pre(context)
{
}

function hf_file(context, phase)
{
}

function hf_post(context)
{
}
```

Important: Only the `hf_file` function is required. The other parts of this structure are optional. For more information, see “Hot Folder Script Functions” on page 311.

Comment and blank lines

Comment lines can appear anywhere in the script file, and are simply ignored. Comment lines start with "//"; that is, with two forward slash characters. Blank lines are meaningless as well, except that a blank line can signal the end of the static parameter section as described in the preceding section.

Hot Folder Script Parameters

A Hot Folder script file starts with zero or more static parameter definitions. A static parameter definition consists of a comment line that has the following structure:

```
// HF_<param name> = <param value>
```

These definitions associate parameter names with parameter values. These values can be referenced by the script code via their names. This mechanism provides a way to allow a script to be clearly configured without an experienced developer having to change values in the code itself.

All script static parameters must be placed at the top of the script file, prior to the first line that does not start with "//". Any parameter definitions that appear after the first non-"//" line is ignored (treated like a simple comment line).

Hot Folder Script Functions

There are three functions used within a Hot folder script:

hf_pre

This function is optional. If it exists, it is called once prior to processing each individual file in a batch. It is passed a handle to the Hot Folder Context object (see “Hot Folder Script Programming” on page 312). The `hf_pre` function can be used to do any initial preparation that is required before files are actually processed.

hf_file

This function is required. It is usually called once for each file to be processed. It is passed a handle to the Hot Folder Context object (see “Hot Folder Script Programming” on page 312) as well as a string indicating the current "Phase." In most cases, the Phase parameter can be ignored and need not even be defined. This function defines the processing that is performed on each file to be processed.

hf_post

This function is optional. If it exists, it is called once after processing each individual file in a batch. It is passed a handle to the Hot Folder Context object (see “Hot Folder Script Programming” on page 312). This function can be used to do any post processing that is required after the files are processed.

Hot Folder Script Programming

The Hot Folder Context object is the programming interface to the Hot Folder system. The code in the three functions above make calls on this object to obtain all the information necessary to process files and control the Hot Folder mechanism.

There are basic features supported by the Hot Folder Context object, which are all that are needed to implement basic Hot Folder operations. There are also some advanced features of the Hot Folder mechanism that can be accessed via this object.

A Simple Hot Folder Script

This Hot Folder script example creates a TIFF thumbnail for each input file. The size of the thumbnail can be changed by modifying the `HF_THUMBNAIL_WIDTH` parameter at the top of the script. The results are written to a per-batch output directory. The `HF_DELETE_ORIGINALS` parameter controls if the original files are deleted or not after processing.

If one or more errors occur, an email message is sent to Joe, Steve, and Jane. If no errors occur, tan email is sent only to Steve and Jane.

```
// HF_ERROR_MAILLIST = joe@somecompany.com
// HF_RESULTS_MAILLIST = steve@somecompany.com, jane@somecompany.com
// HF_THUMBNAIL_WIDTH = 128
// HF_DELETE_ORIGINALS = true

function hf_file(context)
{
// Load the source image
var image = new Media();
image.load(name @ context.getSourcePath());

// Get the thumbnail width parameter value
var width = context.getParameter("HF_THUMBNAIL_WIDTH");

// Make a thumbnail
image.scale(xs @ width, constrain @ true);

// Save the result to the PerBatch output directory. Name the
// resulting file the same as the source file, but with a TIFF extension.
image.save(name @ context.getPerBatchResultPathNoExt() + ".tif");
}
```

```
function hf_post(context)
{
context.setDeleteOriginals(context.getParameter("HF_DELETE_ORIGINALS") == "true");
}
```

Hot Folder File Locations

The function `hf_file` function is a required function and requires both a source and destination location for the processed files.

Retrieving the source file location

The code in the `hf_file` function always requires the location of the source file to process. The full path to the source file is obtained by calling the context `getSourcePath()` method. The following is an example:

```
var sourcePath = context.getSourcePath();
```

Retrieving the destination file location

If the code in the `hf_file` function should write the results of processing a file to one of the standard locations, it can call one of the following methods for the context object:

- `getSharedResultsDir()` - Returns the full path of the Shared output directory.
- `getSharedResultPath()` - Returns the full path to a file in the Shared output directory with the same name as the input file.
- `getSharedResultPathNoExt()` - Returns the full path to a file in the Shared output directory with the same name as the input file, but with the file extension removed.
- `getPerBatchResultsDir()` - Returns the full path of the PerBatch output directory.
- `getPerBatchResultPath()` - Returns the full path to a file in the PerBatch output directory with the same name as the input file.
- `getPerBatchResultPathNoExt()` - Returns the full path to a file in the PerBatch output directory with the same name as the input file, but with the file extension removed.

Accessing Hot Folder Static Parameter Values

The values of the static parameters at the top of a script file can be accessed with the context `getParameter()` method. This method takes a single string parameter that indicates the name of the parameter to retrieve, such as the following example:

```
// HF_WIDTH = 512
…
var width = context.getParameter("HF_WIDTH");
```

Deleting the Original Files

By default, the original files that were copied to the Hot Folder are retained after the job completes. The script can, however, specify that the original files should be deleted after processing is completed. It does this in the `hf_post` function by calling the context `setDeleteOriginals()` method. This method takes a single Boolean parameter that indicates if the originals should be deleted. The following is an example of a complete `hf_post` function that performs this function:

```
function hf_post(context)
{
context.setDeleteOriginals(true);
}
```

Sending Email Messages

Email messages can be sent by a Hot Folder job to indicate success or failure of the job. Two static parameter values placed at the top of the script file determine what email is sent. These static parameters are the following:

- `HF_ERROR_MAILLIST` - A comma separated list of email address to send email to only if errors occur.
- `HF_RESULTS_MAILLIST` - A comma separated list of email addresses to send email to indicating the success or failure of the job.

If one or more errors occur during job processing, email is sent to the addresses on both of the lists described above. This email includes the specifics of each error that occurred.

If no errors occur, email is sent only to the addresses on the `HF_RESULTS_MAILLIST` list.

Advanced Hot Folder Script Programming

Beyond the basic processing, the Hot Folder context object supports some additional methods used for more advanced functionality. You can incorporate these scripting techniques and methods to maximize your processing information and manage your Hot Folder jobs.

Multiple Phase Processing

A single call to the `hf_file` function will always be performed by a single processor. If a single script performs multiple operations on a single input file, and those operations could be performed in parallel, performing those operations in a single call to `hf_file` does not make the best use of available processing resources.

To improve upon this situation and improve a single script's ability to process multiple operations in parallel, a *Phasing* feature available. Phasing causes the `hf_file` function to be called multiple times for a single input file. Each such call can be executed on a separate processor, allowing those calls to be performed in parallel.

To define multiple phases in a script, a static property named `HF_PHASES` is added to the top of the script. The value of this property is a comma separated list of phase names. These names are arbitrary. The `hf_file` function will be called once for each specified phase name. The phase name associated with a particular call to `hf_file` will be passed as the second parameter in that call, allowing the `hf_file` function to differentiate between the different phases.

The following is a simple example of a three-phase script, which writes out the input image in three different output formats. The three save functions can be performed in parallel when processor resources allow:

```
// HF_PHASES = jpeg, tiff, tga
function hf_file(context, phase)
{
    var image = new Media();
    image.load(name @ context.getSourcePath());
    switch (phase)
{
case "jpeg": image.save(name @ context.getPerBatchResultsDir() + "/result.jpg");
break;
case "tiff": image.save(name @ context.getPerBatchResultsDir() + "/result.tif");
break;
case "tga": image.save(name @ context.getPerBatchResultsDir() + "/result.tga");
break;
}
}
```

Hot Folder Logging

The Hot Folder context object provides support for logging of script processing errors and other information.

Errors

Errors are automatically logged when an error occurs in a MediaRich function, such as `load()` or `save()`. It is often the case, however, that the script logic determines that something is wrong and want to specifically log an error. How this is done depends on the desired behavior. If the script should exit at the point of the error, it can simply throw an exception, just as it would in any other MediaRich script. If, however, the script should continue after logging the error, it can use the `logError()` method in the context object. This method takes a single string that is an error message describing the error to be logged.

The following is an example of exiting on an error:

```
if (width > 1024)
{
throw "Thumbnails can be no larger than 1024 pixels wide";
}
```

In this next example, the script recovers after placing a message in the error log:

```
if (width > 1024)
{
```

```
context.logError("Thumbnails can be no larger than 1024 pixels wide");
width = 1024;
}
```

Debugging or other non-error information

It is often desirable to log information to a log file without that information being considered an error. This can be used during script debugging, or to provide information to the Hot Folder user about what was done during processing. A method of the context object named `logInfo()` is provided for this purpose. This method logs to a file named "^Results/<batch ID>_Info", where <batch ID> is replaced with the batch ID of the job.

The following is an example of logging information to the job Info file:

```
context.logInfo("Saving to file " + context.getPerBatchResultPathNoExt() + ".tif");
```

If information is logged in this manner, and email messages are sent by the job, the text of those messages will include the logged information.

Sending Result File Information in Email Messages

It is often desirable to explicitly list the result file locations in the email that is sent by the job. To do this, you could use the `logInfo()` method. As a convenience, a method on the context object named `logResults()` is provided specifically for this purpose. This method takes a single string, which is interpreted as a file path. File paths passed to this function are collected and summarized in the email messages, indicating that these paths are the results that were generated by the Hot Folder job.

Sending Results as Email Attachments

The `logAttachment()` method of the context object allows result files to be sent as attachments in the email messages that are sent by the job. This method takes a single string parameter that must be the path to a file. If the path passed to this method refers to an existing file (often one just generated), then that file is added as an attachment to any emails that are sent on behalf of the job.

Referencing Available Data in the Context Object

The following are a few other methods are available on the context object that retrieve useful information:

- `getBatchId()` - returns the batch ID of the current job
- `getHotFolderPath()` - returns the path of the Hot Folder itself
- `getSourceRelativePath()` - similar to getSourcePath(), but returns a partial path to the source file that is relative to the Hot Folder directory

  This returns only the file name of the source for files that were added directly to the top-level of the Hot Folder (that is, were not in a subdirectory).
- `stoString()` - returns a report of all of the information that the context object contains

  This method is often helpful for debugging.

CHAPTER 9

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 chapter explains the basics of color management and using MediaRich and the MediaScript scripting language to ensure accurate RGB and CMYK color conversions.

Chapter summary

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. 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 colorspacein which an image is represented.

Note: CMYK color values are handled differently than RGB, especially in relation to pad, foreground and background colors, for example. For instance, in CMYK white is represented as 0x00000000, while in RGB it is represented as 0xffffff.

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 could 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 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

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.

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.

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.

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.

IccProfile object

Use the IccProfile object constructor to 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 can be used as a source or destination profile for all color management methods that take profiles as arguments, including the following:

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.

SourceProfile Arguments

The methods `colorCorrect`, `colorFromImage`, `colorToImage`, and `setSourceProfile` all take a SourceProfile argument, a DestProfile argument, or both.

There are three ways to specify 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 (such as 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 the colorspace of one monitor to that of another monitor 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 because 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 is 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 is 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 performs the composite:

```
cmykProfile = new IccProfile(cmykImage);

rgbImage.colorCorrect(SourceProfile @ "rgb",
DestProfile @ cmykProfile, Intent @ "Perceptual");

cmykImage.composite(source @ rgbImage);
```

Note: The `new IccProfile(cmykImage)` method throws 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, add 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 can 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 are 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 is 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. The following is an example of using 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.

Why does a composite() of makeText()-generated CMYK data onto a CMYK background messes up the anti-aliasing?

First, note that use of the CMYK "black" value 0x000000ff is generally not a good choice because it will not appear black in Photoshop or any application with a paper-based color profile. You can produce better results with the following:

```
text.makeText(font @ "arial", text @ "sample", color @ text.colorToImage(color @
0x000000, sourceProfile @ "rgb", destProfile @ "cmyk"), size @ 42, smooth @ true,
rtype @ "CMYKA");
```

This uses the default CMYK profile notion of black (actually more like 0xBFADAAE5). If the destination raster has an embedded profile, it would be even better to use that.

Second, doing a composite with transparencies other than 0 or 255 is usually not a good idea in the CMYK colorspace. This is because most CMYK profiles can not be interpolated linearly as when alpha blending RGB or CMYK. Better to convert to RGB colorspace, do the blend, then convert back to CMYK, since the linear interpolation is generally sufficient for RGB.

CHAPTER 10

MediaRich Metadata Support

MediaRich CORE 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 supports loading XMP metadata from the following file formats: Illustrator, InDesign, EPS, GIF, PDF, and PNG. It also supports IPTC metadata extraction for RAW Camera formats. 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.

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

Chapter summary

Low-Level Metadata Interface

Two MediaScript objects provide support for metadata:

- 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.

Metadata Support in the Media Object

The `load` command of the `Media` object loads and attaches EXIF, IPTC, and XMP metadata if the `loadMetadata` parameter is specified as `true`, such as in the following 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 attaches 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 following code outputs a TIFF and displays the raw XML metadata at the console in Mush mode:

```
#include "sys:/Metadata.ms"
m = new Media();
m.load(name @ "extensis/_DSC1160.NEF", loadMetadata @ true);
m.save(name @ "out/out.tif", saveMetadata @ true);
var meta = m.getMetadata("Exif");
print("Metadata:\n"+meta+"\n");
```

The following code copies a CameraRaw file containing metadata to the Media folder. You can run the script in a browser using this URL:

```
http://localhost/mgen/mediaRichTest.ms?nc=1
```

This demonstrates using XML data representing the Exif data to display in the browser, as well as embedding it in the new image.

```
#include "sys:/TextResponse.ms"
function main()
{
var m = new Media();
m.load(name @ "read:/<a CameraRaw file containing metadata>", loadMetadata @ true);
m.save(name @ "cache:/2767.tif", saveMetadata @ true);
var theXML = m.getMetadata("Exif");
var resultXml = new TextResponse(TextResponse.TypeXml);
resultXml.setText(theXML);
resp.setObject(resultXml, RespType.Streamed);
return;
}
```

Metadata Support in 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, as illustrated in the following example:

```
var metaObj = new _MR_Metadata(“myimage.jpg”, “jpeg”);
```

The file name specifies the file to be processed. The file type parameter indicates the type of that file. If the file has a valid extension, the file type can be omitted.

The `_MR_Metadata` object `load` command extracts one type of metadata from the image as an XML document. It takes a single string parameter indicating which type of metadata to extract. The three valid values for the parameter are `EXIF`, `IPTC` and `XMP`. The following is an example:

```
var myExifDoc = metaObj.load(“Exif”);
```

If the file has a valid extension, the file type can be omitted.

The `_MR_Metadata` object `save` command embeds one or more types of metadata within the image file. The parameters for the `save` command object are `exif`, `iptc`, `xmp`, and `name`. The file type of the saved file 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 is transferred to the output file. If any of the `exif`, `iptc`, or `xmp` parameters is specified as `null`, existing metadata of that type is omitted in the output. Otherwise, IPTC and XMP data is replaced using the specified data, and writable `EXIF` tags are replaced.

Note: The `EXIF` camera data tags are never replaced.

The following code will dump metadata as XML:

```
var data = new _MR_Metadata("tbimages:/metadata/Canada_021.eps",
"eps");
var xmp = data.load("XMP");
print(xmp + "\n");
```

You can also put "EXIF" and "IPT" in the `load()` call to get those data types.

High-Level Support for EXIF, IPTC, and XMP

There are two MediaScript objects 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 and provides `set<Tag>` methods and `get<Tag>` methods, which set and get individual metadata fields. `<Tag>` represents the name of the metadata tag to set or get.

One or more `#include` lines are required in scripts that call metadata functions:

- For EXIF support: `#include “sys:/ExifMetadata.ms”`
- For IPTC support: `#include “sys:/IPTCMetadata.ms”`
- For XMP support: `#include “sys:/XMPMetadata.ms”`

The following sections describe 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.

IPTC Metadata Object

Use the following methods to retrieve and set metadata values for IPTC metadata. Refer the schema (*IPTC.xsd*) in the */Shared/Originals/Sys* folder for the required format for each of the IPTC fields.

Note: The information provided here includes only a brief description. For a complete description of each IPTC metadata field, consult the IPTC news-photo metadata specification available at http://www.iptc.org under the title “Digital Newsphoto Parameter Record.”

For the following methods, 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

Method	Description
`setObjectTypeReference (string)`	Sets the object type reference field
`getObjectAttribute Reference()`	Returns an array of attribute references
`setObjectAttributeReference (string, ...)`	Sets attribute references
`addObjectAttributeReference (string, ...)`	Adds an attribute references to the list
`setObjectAttributeReference Array(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
`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

Method	Description
`addSupplementalCategory (string, ...)`	Adds a value to the supplemental category list
`setSupplementalCategory Array(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]
`getContentLocationCode (which)`	Returns the `ContentLocationCode` subfield of the `ContentLocation` tag indexed
`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
`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
`setReference(service, date, number)`	Sets the Reference element to the reference specified by service, date, and number **Note:** `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. **Note:** `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()`	Returns the `OriginatingProgram` field
`setOriginatingProgram (string)`	Sets the `OriginatingProgram` field

Method	Description
`getProgramVersion()`	Returns the `ProgramVersion` field
`setProgramVersion(string)`	Sets the `ProgramVersion` field
`getObjectCycle()`	Returns the `ObjectCycle` field
`setObjectCycle(string)`	Sets the `ObjectCycle` field
`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
`getByLineTitle(which)`	Returns the `ByLineTitle` property of the ByLine element specified
`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`
`getOriginalTransmission Reference()`	Returns the `OriginalTransmissionReference`
`setOriginalTransmission Reference(string)`	Sets the `OriginalTransmissionReference`
`getHeadline()`	Returns the `Headline`
`setHeadline(string)`	Sets the `Headline`
`getCredit()`	Returns the `Credit` field

Method	Description
`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
`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`
`setLanguage Identifier(string)`	Sets the `LanguageIdentifier`

The EXIF Metadata Object

The following methods may be used to get and set metadata values for EXIF metadat`a. Please refer to 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 for the following methods 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
`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
`getPhotometric Interpretation()`	Returns the photometric interpretation
`getPlanarConfiguration()`	Returns the planar configuration
`getYCbCrSubSampling()`	Returns the YCbCr sub-sampling

Method	Description
`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
`getPrimary Chromaticities()`	Returns the primary chromaticities
`getYCbCrCoefficients()`	Returns the YCbCr coefficients
`getReferenceBlack White()`	Returns the ReferenceBlackWhite value

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
`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

Method	Description
`getImageLength()`	Returns the image height
`getBitsPerSample()`	Returns the number of bits per sample
`getCompression()`	Returns the compression type
`getPhotometric Interpretation()`	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
`getPrimary Chromaticities()`	Returns the primary chromaticities
`getYCbCrCoefficients()`	Returns the YCbCr coefficients
`getReferenceBlack White()`	Returns the ReferenceBlackWhite value

IFDEXIF Methods

Method	Description
`getVersion()`	Returns the EXIF version
`setVersion(string)`	Sets the EXIF version (default is `2.2`)
`getUserComment()`	Returns the user comment
`setUserComment(string)`	Sets the user comment
`getColorspace()`	Returns the colorspace
`getPixelXDimension()`	Returns the width
`getPixelYDimension()`	Returns the height

Method	Description
`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

Method	Description
`getFocalPlaneYResolution()`	Returns the focal-plane vertical resolution
`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
`getSubjectArea()`	Returns the subject area.
`getMakerNote()`	Returns the manufacturer notes.
`getCustomRendered()`	Returns the custom image procesing.
`getExposureMode()`	Returns the exposure mode.
`getWhiteBalance()`	Returns the white balance.
`getDigitalZoomRatio()`	Returns the digital zoom ratio.
`getFocalLengthIn35mmFilm()`	Returns the focal length in 35mm film.
`getSceneCaptureType()`	Returns the scene capture type.
`getGainControl()`	Returns the gain control.
`getContrast()`	Returns the contrast.
`getSaturation()`	Returns the saturation.
`getSharpness()`	Returns the sharpness.
`getDeviceSettingDescription()`	Returns the device settings description.
`getSubjectDistanceRange()`	Returns the subject distance range.
`getImageUniqueID()`	Returns the unique image ID.

EXIF GPS Metadata Extraction

MediaRich 4.0 includes new methods for extracting GPS metadata from the `ExifMetadata` object.

Method	Description
`getGPSVersionID()`	Returns the GPS version ID
`getGPSLatitude()`	Returns the GPS Latitude
`getGPSLongitude()`	Returns the GPS Longitude
`getGPSAltitude()`	Returns the GPS altitude
`getGPSTimeStamp()`	Returns the GPS time stamp
`getGPSDateStamp()`	Returns the GPS date stamp
`getGPSSatellites()`	Returns the GPS satellite information
`getGPSStatus()`	Returns the current GPS status
`getGPSMeasureMode()`	Returns the GPS measurement mode
`getGPSDOP()`	Returns the GPS data degree of precision (DOP)
`getGPSSpeed()`	Returns the GPS speed
`getGPSTrack()`	Returns the GPS track
`getGPSImgDirection()`	Returns the GPS image direction
`getGPSMapDatum()`	Returns the geodetic survey data used by the GPS receiver
`getGPSDestLatitude()`	Returns the latitude of the GPS destination point
`getGPSDestLongitude()`	Returns the longitude of the GPS destination point
`getGPSDestBearing()`	Returns the bearing to the GPS destination point
`getGPSDestDistance()`	Returns the distance to the GPS destination
`getGPSDifferential()`	Returns value to indicate if differential correction is applied to the GPS receiver

These methods do not take any parameters. The returned values are converted from raw form to a human-readable friendly form, suitable for direct display to the user.

The following is an example script that demonstrates these GPS metadata methods:

```
var exif = new ExifMetadata(false);
exif.loadFromFile("GPSExample.jpg");
print("GPS Version ID: "+exif.getGPSVersionID()+"\n");
print("GPS Latitude: "+exif.getGPSLatitude()+"\n");
print("GPS Longitude: "+exif.getGPSLongitude()+"\n");
print("GPS Altitude: "+exif.getGPSAltitude()+"\n");
print("getGPSTimeStamp: "+exif.getGPSTimeStamp()+"\n");
print("getGPSDateStamp: "+exif.getGPSDateStamp()+"\n");
print("getGPSSatellites: "+exif.getGPSSatellites()+"\n");
print("getGPSStatus: "+exif.getGPSStatus()+"\n");
print("getGPSMeasureMode: "+exif.getGPSMeasureMode()+"\n");
print("getGPSDOP: "+exif.getGPSDOP()+"\n");
print("getGPSSpeed: "+exif.getGPSSpeed()+"\n");
print("getGPSTrack: "+exif.getGPSTrack()+"\n");
print("getGPSImgDirection: "+exif.getGPSImgDirection()+"\n");
print("getGPSMapDatum: "+exif.getGPSMapDatum()+"\n");
print("getGPSDestLatitude: "+exif.getGPSDestLatitude()+"\n");
print("getGPSDestLongitude: "+exif.getGPSDestLongitude()+"\n");
print("getGPSDestBearing: "+exif.getGPSDestBearing()+"\n");
print("getGPSDestDistance: "+exif.getGPSDestDistance()+"\n");
print("getGPSDifferential: "+exif.getGPSDifferential()+"\n");
```

Common Metadata Methods

The IPTCMetadata and EXIFMetadata objects have several common methods allowing the script developer 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)`	Loads metadata object with data from the specified image file
`loadFromMedia(media)`	Loads metadata object with data from the specified media
`loadFromXML(xmlString)`	Loads metadata object with data from the specified XML string
`blankDocument()`	Loads metadata object with a valid blank document
`validate()`	Validates the document to the appropriate schema
`isEmpty()`	Returns `true` if the document is empty
`XMPMetadata(validate)`	Creates a blank XMP metadata document

Method	Description
`loadFromDiskFile(filename)`	Loads an XMP document from the named disk file to support "sidecar" XMP documents
`getSingleValueForNode(node)`	Retrieves a single value for the specified node If there are enclosed sub-elements, their node values are concatenated with newlines.
`getTagValue(tagName)`	Retrieves a single value for the element tagName rdf:Bag and rdf:Seq element values are returned as a single string with newlines separating the individual values. The tag should use a "standard" namespace alias (see "XMPMetadata Namespaces" on page 343 for values). Before searching the document, tagName is mapped into the documents namespace alias space from the "standard" space.
`setTagValue(value, tagName)`	Set s the value for the simple element designated by tagName The tagName must use one of the namespace aliases defined in XMPMetadataBaseNSTable.
`setTagBagValue(tagName, value)`	Sets a multi-valued element using the rdf:Bag construct The value must be either an array or a newline delimited string. The tagName must use one of the namespace aliases defined in XMPMetadataBaseNSTable.
`setTagSeqValue(tagName, value)`	Sets a multi-valued element using the rdf:Seq construct The value must be either an array or a newline delimited string. The tagName must use one of the namespace aliases defined in XMPMetadataBaseNSTable.

Note: Each metadata constructor creates a blank metadata document of the appropriate type. The document is not empty, but is a valid XML document for the appropriate metadata schema. However, `loadFromFile` and `loadFromMedia` 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 return `null` if the document is empty.

Enabling and Disabling Exif Validation

There are times when you want to catch images with badly formed Exif information, perhaps to ensure that the files you send out contain fully valid Exif. To do this, ensure that validation is on, such as the following example:

```
var exif = new ExifMetadata();
exif.loadFromXML(exifXml);
```

Other times, you might want validation to be off. For example, you could write a script that manipulates one of the Exif fields, where you do the validation on just one field in the script rather than requiring that the entire Exif blob is valid.

```
var exif = new ExifMetadata(false);
exif.loadFromXML(exifXml);
```

XMPMetadata Namespaces

The namespaces currently defined in the XMPMetadataBaseNSTable are as follows:

```
XMPMetadataBaseNSTable["http://www.w3.org/1999/02/22-rdf-syntax-ns#"] = "rdf";
XMPMetadataBaseNSTable["http://iptc.org/std/Iptc4xmpCore/1.0/xmlns/"] =
"Iptc4xmpCore";
XMPMetadataBaseNSTable["http://ns.adobe.com/exif/1.0/"] = "exif";
XMPMetadataBaseNSTable["http://ns.adobe.com/photoshop/1.0/"] = "photoshop";
XMPMetadataBaseNSTable["http://ns.adobe.com/xap/1.0/"] = "xap";
XMPMetadataBaseNSTable["http://ns.adobe.com/xap/1.0/rights/"] = "xapRights";
XMPMetadataBaseNSTable["http://purl.org/dc/elements/1.1/"]= "dc";
XMPMetadataBaseNSTable["http://ns.adobe.com/xap/1.0/sType/Job#"]= "stJob";
XMPMetadataBaseNSTable["http://ns.adobe.com/xap/1.0/bj/"]= "xapBJ";
XMPMetadataBaseNSTable["http://ns.adobe.com/xap/1.0/mm/"]= "xapMM";
XMPMetadataBaseNSTable["http://ns.adobe.com/tiff/1.0/"]= "tiff";
XMPMetadataBaseNSTable["http://ns.adobe.com/pdf/1.3/"] = "pdf";
XMPMetadataBaseNSTable["http://ns.adobe.com/iX/1.0/"] = "iX";
XMPMetadataBaseNSTable["http://ns.adobe.com/xap/1.0/sType/ResourceRef#"]= "stRef";
XMPMetadataBaseNSTable["http://ns.adobe.com/camera-raw-settings/1.0/"] = "crs";
XMPMetadataBaseNSTable["http://ns.adobe.com/exif/1.0/aux/"] = "aux";
```

When naming an element in an XMP document, you must use namespaces from this list, regardless of the actual namespace alias used in the document. When an XMP document is opened, any namespace aliases in the document are mapped to values in this list using the namespace URLs.

You can add your own namespace aliases to the list in the `Shared/Originals/Sys/XMPMetadata.ms` file.

Custom SWF XMP Support

The following is an example of high-level Metadata API extraction:

```
var xmp = new XMPMetadata(false);
xmp.loadFromFile("Metatest.swf");
print("Title: "+xmp.getTagValue("dc:title")+"\n");
print("Description: "+xmp.getTagValue("dc:description")+"\n");
```

The following is an example of SWF low-level API extraction (dumps the raw XML):

```
var meta = new _MR_Metadata("metatest.swf");
var xml = meta.load("XMP");
print(xml + "\n");
```

The regular SWF metadata is retrieved via `Media.getFileInfo()` and includes the following:

- Width
- Height
- Frames
- FrameRate
- Format
- XDpi
- YDpi
- ResolutionUnits
- XResolution
- YResolution
- Version
- Type

Any other metadata must be retrieved using the MediaRich XMP metadata API.

Note: The SWF reader supports up to Flash 7, with some support for Flash 8,9, and 10.

SWF reading works like other multpage/multiframe readers. You can specify the frame or frames you want to load into the media object with the frame or frames param used with `Media.load()`. Ranges such as "4-10", for example, are valid as well as specific frame numbers, such as "4,8,20".

You can then operate on the frames as you would with any multipage/multiframe media object, using `Media.getFrame()`.

Also, a "time" parameter can also be used. You can specify how many seconds into the movie to return a single frame. The time parameter always returns a frame from readable movies. Requesting a specific frame could fail if it is beyond the number of frames in the movie or past user input sections.

`Media.load()` also accepts the Xs and Ys parameter(s) for SWF which force the frame to be rendered at a specified size. If they are left off, the default size stored in the SWF file is used.

The detect parameter also works with SWF, so files that lack the ".swf" extension can still be loaded.

The frames have no times present so if you want to convert a short sequence of frames into a GIF animation, you will need to specify the frame rate during the save.

Note: Extracting multiple frames causes a temp image file to be created for each frame on disk, so do not try to convert entire movies this way since it will take a very long time and may cause disk full errors.

It is NOT possible to seek in SWF files—if you specify frames that are deep into the movie, the extraction could take a very long time because the movie must be played back sequentially internally.

Examples

```
var m = new Media();
m.load(name @ "Example.swf", time @ 10);
m.save(name @ "out.tif");
m.load(name @ "Example.swf", frame @ 45);
m.save(name @ "out.jpg");
m.load(name @ "Example.swf", frames @ "0,10,20");
m.save(name @ "out.tif");
m.load(name @ "Example.swf", frames @ "7-12");
m.save(name @ "out.tif");
```

XMP Universal “blob” Find and Replace

The need to provide a “universal XMP embedder” to enable a wide range of potential file formats for XMP metadata embedding arose in 2009 for Equilibrium’s OEMs. This MediaScript object searches any file given to it for an XMP blob and replaces that metadata with the new XMP metadata blob that is created or extracted. It is not capable of adding new XMP blobs to a file.

Note: Any file type that can support XMP metadata blobs can be updated. Refer to Adobe's XMP specification for more information about supported XMP filetypes.

The plugin adds the `XMPEmbed` object to MediaScript when the `XMPEmbed.mdv` is linked to the script. The object has only a constructor and one method - `embed()`.

- The `new XMPEmbed()` constructor takes the file path for the file to update as the only parameter. The file must be writable and it must exist.
- The `embed()` method takes only one parameter - a string object that contains the XMP metadata to embed.

No parsing or validation of the data is done, so it is entirely possible to put XMP data from one format into another and confuse applications that read the XMP metadata.

Important: It is strongly recommended that you work on a backup copy of the file because the data is changed in-place and if something goes wrong, the original data cannot be recovered. However, most errors (except for a physical write failure to the medium) are caught before any actual data is replaced.

The following sample code demonstrates this use:

```
// Replace XMP metadata in one file with data from another file
#link "XMPEmbed.mdv"
var meta = new _MR_Metadata("SourceFile.eps");
var xml = meta.load("XMP");
var xmpWriter = new XMPEmbed("DestFile.eps");
xmpWriter.embed(xml);
```

Metadata from AVLowLevel API

Important: As QuickTime's professional transcoding capability is deprecated, the MediaRich CORE A/V 1.1 is also discontinued. We highly recommend that you do NOT utilize the MediaRich CORE 1.1 API unless absolutely necessary for a specific operation. Additionally, A/V CORE 1.1 is 32-bit only, so you will need to retrieve the 32-bit MediaRich 4.0 before proceeding with any A/V CORE 1.1 usage.

There are three Java classes in the AVLowLevel Java API that allow access to metadata. QTUserData (part of the QuickTime handler) accesses metadata for QuickTime video files, QTWMVIWMHeaderInfo3 (available only on the Windows platform) accesses metadata for Windows Media video files, and AVElement (part of the ffmpeg handler) allows access to other audio/video metadata. For information about what file formats each handler (QuickTime, Windows Media, ffmpeg) is assigned to handle, see "Adding QuickTime Support" on page 269.

The documentation for the AVLowLevel API, which documents the QTUserData, QTWMVIWMHeaderInfo3, and AVElement classes, is located at *Equilibrium/MediaRich All Media Server/Mediarich Documentation*. On Windows system, you can access the AVLowLevel Java API documentation from the Start menu.

The AVElement class has the getMetaData method, which return metadata such as title, author, copyright, comment, album, genre, year, and/or track. Here is an example.

```
var element = new AVElement("SomeMovie.mpg", 1, AVElement.AnyStream);
var metadata = element.getMetaData();
print("The title is " + metadata.title + "\n");
```

The AVLowLevelDocs documentation for the QuickTime class contains a list of many pre-defined constants that you can use to access QuickTime metadata. When you use these constants, you must precede the constant name with "QuickTime.", such as QuickTime.codecLowQuality, since these constants are members of the QuickTime class. The constants listed in the AVLowLevelDocs for QuickTime metadata are:

```
Quicktime.kUserDataIPTC
Quicktime.kUserDataMovieControllerType
Quicktime.kUserDataName
Quicktime.kUserDataTextAlbum
Quicktime.kUserDataTextArtist
Quicktime.kUserDataTextAuthor
Quicktime.kUserDataTextChapter
Quicktime.kUserDataTextComment
Quicktime.kUserDataTextComposer
Quicktime.kUserDataTextCopyright
Quicktime.kUserDataTextCreationDate
Quicktime.kUserDataTextDescription
Quicktime.kUserDataTextDirector
Quicktime.kUserDataTextDisclaimer
Quicktime.kUserDataTextEditDate1
Quicktime.kUserDataTextEncodedBy
Quicktime.kUserDataTextFullName
Quicktime.kUserDataTextGenre
Quicktime.kUserDataTextHostComputer
Quicktime.kUserDataTextInformation
Quicktime.kUserDataTextKeywords
Quicktime.kUserDataTextMake
Quicktime.kUserDataTextModel
Quicktime.kUserDataTextOriginalArtist
Quicktime.kUserDataTextOriginalFormat
Quicktime.kUserDataTextOriginalSource
Quicktime.kUserDataTextPerformers
Quicktime.kUserDataTextProducer
Quicktime.kUserDataTextProduct
Quicktime.kUserDataTextSoftware
Quicktime.kUserDataTextSpecialPlaybackRequirements
Quicktime.kUserDataTextTrack
Quicktime.kUserDataTextURLLink
Quicktime.kUserDataTextWarning
Quicktime.kUserDataTextWriter
```

The QTWMVIWMHeaderInfo3 class methods use strings to access the metadata. Here is a sample call.

```
var info = new QTWMVIWMHeaderInfo3("dixiechicks_theresyourtrouble_120.wmv");
// Show all the file level attributes
var count = info.getAttributeCountEx(0);
for (i = 0 ; i < count ; i++)
{
var attr = info.getAttributeByIndexEx(0, i);
print("name: " + attr.name + " value: " + attr.value + "\n");
}
```

Some of the more common tag strings supported by Windows Media files include the following:

```
Duration
Bitrate
Seekable
Stridable
Broadcast
Is_Protected
Is_Trusted
Signature_Name
HasAudio
HasImage
HasScript
HasVideo
CurrentBitrate
OptimalBitrate
HasAttachedImages
Can_Skip_Backward
Can_Skip_Forward
FileSize
Title
Author
Copyright
Description
Rating
WMFSDKVersion
WMFSDKNeeded
IsVBR
```

APPENDIX A

MediaRich Programming Best Practices

MediaRich is a very flexible image serving platform. There are many ways to configure and utilize its features. This appendix provides some guidelines for making the best use of MediaRich features for creating, distributing, and managing your media assets.

Refer to the *MediaRich CORE Installation and Administration Guide* for information about best practices for installation and configuration.

Appendix summary

Media Creation Best Practices

When implementing any MediaRich solution, it is important that guidelines are established for the creation of the images. These guidelines will help minimize the confusion between the MediaScript developer and the graphic designers.

Workflow

The general workflow is that the graphic designers work with their design tools, such as Photoshop, to create the original images. The Web developers work with MediaScript to create the templates for the media. The original images and the scripts are placed on the MediaRich server during a push process. Upon request for an image from a Web user, MediaRich generates the Web-ready image and serves it to the browser.

File naming

It is important to adopt a standard file naming convention so that designers and developers can quickly locate files. Some typical name conventions utilize SKU, product ID, or product name as the name of the image. Another strategy is to create a catalog of images and store the information about the images in a database, text file, or spreadsheet. The catalog could contain the file name, location of the file, latest revision, and description of the image. The catalog should be stored in a central location so the team can share it.

Masks

If an image is used for colorization or compositing, it is important that the image contain a mask. MediaRich does not require masks and can use GIF or PNG transparency information if it is available. If the image does not contain a mask, the graphic designer must create a mask using an image-editing program. When the mask is created, save the file in a format that preserves the mask information, such as PSD, Targa, or TIFF. The image can then be used for compositing or colorization using MediaRich commands. If the image exists in a Photoshop layer, it is also possible to extract the layer and use it for compositing or colorization. Masks can also be created on the fly in MediaRich by using a `selection()` operation.

Photoshop layers

MediaRich can also collapse Photoshop layers. One use of this feature is to add mix and match functionality. For example, a graphic designer can create all possible combinations of office furniture and store them in a Photoshop file. As long as the Photoshop layers have specific names, it is possible to have the end user call a script that collapses the specified layers which generates a specific combination of furniture images. Individual Photoshop layers can be extracted at any time and used for compositing. If you want to use the whole Photoshop file, you can automatically collapse the Photoshop file during the `load` command.

Colorization

The colorization of an object, such as a sweater ,can be done in a couple of different ways. One method is to select the area to colorize using a mask, then apply the colorize operation. This is the preferred method. RGB values for the different color combinations should be kept in a database, XML, or text file. This way an external process can control the colorization of the object. The extreme colors of solid black and solid white 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.

Another method is to build up a Photoshop file with multiple layers, each containing a different color combination. A MediaScript can be created that will collapse the specified layer. This method allows the artist to have more control, but is labor intensive.

Localized text

MediaRich supports 16-bit Unicode. It is possible to generate localized text in the form of GIF or JPEG file. By storing the localized text in a database or text file it is easy to generate localized text. This allows for the creation of navigation that can be driven by data in a database.

Mix and match

Mix and match functionality can be accomplished either through Photoshop layers or by compositing images. If you wish to use Photoshop layers place each product in a Photoshop layer and use Photoshop layer tools to position the object correctly. You can create a simple MediaScript that collapses the layers as specified. For more information about collapsing layers, see “collapse()” on page 84.

Zoom Resolution

If an image is to be used for zooming or panning it is important that a high-resolution image be used as the source image. A good rule of thumb is to zoom on an image that is at least 1.5 times the size of your zoom window. If your zoom window is 200 x 200 pixels than make sure your source image is at least 300 x 300 pixels. Of course, if you wish to have a high quality zoom experience the larger the source files the better.

Zooming can be arbitrary or based on pre-defined grids. Grid-based zooming limits the possible combinations and can be pre-cached. Arbitrary zooming on any region can have an infinite number of possibilities and is therefore more dynamic. Grid-based zooming is most useful for retail applications. Arbitrary zooming is good for photographs or maps.

MediaScript Integration

MediaScript is the scripting language for MediaRich. It is 100% ECMA compliant. For more guidelines on the language, refer to any resource that covers Jscript, JavaScript, or ECMA Script. The complete language specification can be found at http://www.ecma-international.org/.

XML files

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, refer to https://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/.

XML can be used to exchange data with other external systems such as application servers and databases. As an example, you can use XML to retrieve metadata about an image, such as file location, file description, available colors, crop coordinates, pricing information, and localizable text.

Text data files

MediaScript can also read and write text files. Text files can be used to store and retrieve information. Text files can be used as another method of integrating MediaRich with other systems. One part of your system could generate text files that contain product information including image file name, size, and price. A MediaScript could take the text file as a parameter. The MediaScript would parse the text file to determine the image file name, size, and price and generate the image.

Managing MRL Parameters

You can pass as many parameters you want on a URL. However, because some browsers will not recognize long URLs, it is a best practice to limit the MRL length to below 1024 characters. The order of the parameters is determined by the parameter ordering in the script. It is up to you to determine the parameters you want to pass to the script.

Embedding parameters in the script file

One strategy is to minimize the parameters is to add them to the script. This keeps the MRL simple. Additional parameters could be hard coded as local variables in your script.

Adding parameters to data files

Another strategy is to create a text or XML file that contains the parameters and pass it to the script. This hides all data from the URL. The script would parse the data file and use the information to generate the image.

Appending user profile/device profile

User profile information adjusts image color and quality for a specific viewing device based on the device quality and bit depth. By appending the user profile information to a MRL, the final image can be adjusted to match the output device. A profile script handles the default behavior for different types of devices. The profile script can be modified to change the behavior or add more device support.

Setting the TTL (time-to-live)

You can add an additional parameter to a MRL that will set the TTL of an image. The TTL will tell MediaRich when the image should be flushed from its cache. The TTL will also work with upstream caches.

Managing Performance

There are some best practices you can implement to improve MediaRich performance.

Note: MediaRich is 100% compatible with Akamai. Equilibrium is a certified partner of Akamai and has been tested with the Akamai network. Akamai ARLs can be prepended to a MRL.

Pre-caching

It is sometimes desirable to pre-cache all images on the MediaRich server. The advantage to this is to reduce any possible image generation overhead. There are many ways to pre-cache an image. One method is to use a Perl script, VB script, or a Web crawler to request an image link. This can be done on the production servers or in staging. If it is done in staging then copy the cache directory to the live servers.

Dynamic functionality

The biggest performance hit any image server will encounter is when an image needs to be dynamically generated. The amount of time that is required to open an original image, perform some image processing, and save to a Web-safe image format is far greater than serving a generated image. The best strategy is to try to limit the amount of generations that need be performed on a live server. One way to accomplish this is to pre-cache the images before they are served to the public. Another method is to limit the dynamic nature of your Web site. A good ratio is to have MediaRich serve 90% cached and 10% dynamic images. All images served by MediaRich can be pre-cached including zoomed images, mix-and-match imagery, and colorized images.

APPENDIX B

MediaScript Troubleshooting

If there is a critical error (such as the file or variable name is misspelled and therefore cannot be found), MediaScript cannot execute and you will see a message explaining the error.

If there is a non-critical error, MediaScript could still execute, using default values in place of the erroneous commands or parameters.

Appendix summary

MediaScript Problems

The following table describes typical MediaScript problems and their solutions:

Problem	Solution

Using `importChannel()` to import an alpha mask returns a `No such gun in this raster` error message.	To import an alpha mask, the source image must be 32 or 40-bit (capable of supporting an alpha channel).
	To fix this error, add the following line to your script before the `importChannel()` function:
	`<media_object>.convert(rtype @ “32- bit”);`
	or
	`<media_object>.importChannel(rtype @ “32-bit”);`

Using `load()` fails to load an image and returns a `FlySDK failed to render the file` error message.	If the file has no inherent corruptions, this error can occur when there is insufficient memory to rasterize the image at the MediaRich CORE default of 150 DPI. There are two ways to address this: • Specify a smaller DPI in the load() method • Increase the system memory **Note:** This error can happen with Illustrator files, both bitmap and vector Encapsulated PostScript files, plain PostScript (.ps) files, PDF files, and all Office file types.
Some layers in a Photoshop file cannot be accessed.	Make sure all the layers in the Photoshop file are visible when you save it. Open and resave the file if necessary. Alternatively, call the `setLayerEnabled()` function to make the invisible layers visible. Also, effects layers are ignored by MediaRich. To make them accessible, transform them into regular layers in Photoshop.
MediaRich doesn't recognize EPS paths in Photoshop files.	MediaRich doesn't have a vector engine and it does not parse Postscript files. To solve this problem, transform the EPS path to an alpha channel or layer mask in Photoshop.

Problem	Solution
Using `getLayerIndex ("background")`, or `collapse(track @ "background")` on a Photoshop file with a background layer, returns an error message that the layer or track was not found.	MediaRich can access Photoshop layers by either layer name or index number. A background layer, however, is a special layer type and background is not a proper layer name. Examples: `collapse(track @ "0")` or `getLayerIndex ("0")`.
The `collapse()` function in MediaRich delivers a different result than the Photoshop equivalent.	MediaRich handles layers differently than Photoshop. To apply the Photoshop standard, set the `likePS` parameter in the `collapse()` function to `true`. Example: `collapse(likePS @ true)`. Setting this option attempts to duplicate Photoshop results.

Script Errors

Script errors all start with the MRL on the first line, followed by the line:

```
An exception was thrown in <function name> near line <line number> of script "<script
name>":
```

Use `<script name>`, `<function name>`, and `<line number>` to locate where the error occurred.

The following table lists error messages that you may encounter in the *ScriptErrors.log* file and their probable causes.

Note: This information assumes that MediaRich is properly installed and the different logs are written to the correct locations.

Error	Probable cause
`TypeError 1406: Variable is not a function type.`	This exception indicates that an attempt was made to call an invalid method of an object, or to use a variable as a function. `<function name>` is the name of the function in which the error occurred. `<line>` is the line number in the script file `<script>`. If this error occurs with the MediaRich test shortcut, then either the `ProgramPath` variable in the Local Properties is set incorrectly, or the TextGDI pipe device is missing or invalid.
`SourceError 0001: Unable to open source file "sys:/executeScript.ms" for reading.`	The `SysPath` variable in the Local Properties file is invalid. This variable must point to the location of the *executeScript.ms* script file.

Error	Probable cause
`Could not find function '<name>'.`	The function designated on the MRL as `f=<name>` does not exist in the script. Check case of the name.
`A filter error occurred while processing '<mrl>':` `Missing value for <mrl> at <location>`	Two ampersands in a row in the MRL.
`Illegal argument <arg> at position 1 of (<arg>).`	The argument `<arg>` specified on the MRL is not valid. Check the syntax and encoding, or enclose the argument in quotes.
`'Illegal argument to '<method>' method.'`	An argument was passed to a Media object that did not use the `@` notation, or it was not an object.
`SyntaxError 1410: All parameters must be passed by name if any are.`	Some arguments passed to a Media object did not use the `@` notation, while other arguments did.
`'MediaScript timeout after <n> seconds'`	The script ran for longer than `<n>` seconds. `<n>` is specified in the `MaximumExecutionTime` global property.
`'No format found for file'`	The file type specified in the save method, or the extension given in the load command does not correspond to a known file format.
`'File function not supported'`	Internal file I/O error indicating that a requested file operation (for example, `save()`) is not supported for the given file type.
`'Color Map Too Big'`	The color palette is greater than the maximum size (currently used only by Targa reader, with a set maximum of 256 colors).
`'Can’t Open File'`	MediaRich cannot create the file in write mode. Check file permissions.
`'Disk Full'`	The disk is full and MediaRich cannot write any more data to the file.
`'Digimarc: [digimarc error]'`	Indicates an error in the Digimarc library.
`'File Mangled'`	The file data did not match expected data. Check to see if the file is corrupted.
`'File not found'`	The file does not exist in the specified location. Check spelling and location.
`'File too short'`	More data was expected in the file. Check to see if the file is corrupted.
`'File Type Wrong'`	The file is not the expected type.

Error	Probable cause
`'Format Unsupported'`	The raster type is not supported by the specified file format.
`'Frames different size'`	The frames in a multi-frame Media object (for example, GIF animation) are different sizes.
`'Couldn't find the requested font'`	The font requested could not be found.
`'Insufficient Memory'`	The system did not have sufficient memory to process the request.
`'No such gun in this raster'`	The requested channel type (such as the red or alpha channel) is not present in the raster type.
`'Parameter Bad'`	An invalid parameter was passed to a Media method.
`'Parameter Missing'`	The script does not specify a parameter required by the media object function.
`'Wrong raster type'`	The operation does not support the selected raster type.
`'No media for function'`	An operation was attempted on an empty Media object.
`'Operation requires license'`	Indicates that a separate license is required to perform the requested operation. Contact Equilibrium customer service.
`'PIFF hunk not found'`	An internal file I/O error indicates unexpected or missing data.
`'Parameters clash'`	The script specifies incompatible parameters. For example, you have specified values for both the `Color` and `Index` parameters in the `glow()` function.
`'File already exists'`	An attempt was made to write over a locked file.
`'No appropriate file system was found'`	The virtual file system requested is undefined.
`'You cannot use '..' in file paths'`	Enclosing directories are not accessible using the `..` notation. Specify the path.
`'An attempt was made to write to or change a read-only filesystem'`	The virtual filesystem for a save is designated read-only.
`'A reference was made to an undefined filesystem'`	The filesystem specified in a read or write request is undefined. See Installing and Managing MediaRich.
`'The attempted operation does not allow mixing RGB and CMYK image colorspaces'`	Operations such as collapse, require that all components (layers or frames) of a media be in the same colorspace (i.e., RGB or CMYK).

Error	Probable cause
`'The attempted operation is not supported on CMYK images'`	The operation cannot be performed on CMYK images.
`'The Color Management Plugin is not installed'`	An operation requiring the Color management plugin (*ColorManage.pdv*) was attempted and the device is not installed. Check your installation.

MediaGenerator Errors

If you experience issues for loading multi-page documents or other MediaGenerator errors, be sure to review your configuration to ensure that it meets the following guidelines:

For additional information about MediaRich Server configuration, see the *MediaRich CORE Installation and Administration Guide* .

Windows installations

Minimum recommended configuration for imaging, documents, office applications (no heavy video transcoding, 4k or UHD video transcoding):

- Dual 3+ GHz Intel® Xeon Sandy Bridge or later processors
- 8 GB memory
- Microsoft Windows Server 2012 or later
- Microsoft Internet Information Services® (IIS) and ASP.NET with sub-components

Minimum recommended configuration when processing large HQ videos such as 4k, UHD or 1080P, images, multi-page documents (bare metal highly recommended for video transcoding!):

- 8 Core Intel Xeon Sandy Bridge or later processors
- Dedicated MediaRich Server 8 Core License in shared cache configuration
- Windows Server 2012R2 or later installed (2012R2 is required for GPU Option)
- Dedicated cache volume 4 x 250 GB Disk 15K SAS RAID-10
- OS-Volume 2x150 GB/SAS/10K mirror
- 32 GB Ram

Loading Raw Camera Files

The variety of raw camera file formats in the world is staggering. MediaRich can read many of them. However, because many of these formats do not have consistent file extensions and some of the extensions on these files conflict with other file types, MediaRich limits the number of extensions it associates with raw camera files. As of this writing, the file extensions recognized by MediaRich as raw camera files are the following:

> .dng, crw, .dcr, .kdc, .k25, .dc2, .orf, .raf, .raw, .rw2, .nef, .cr2, .mos, .cs1, .arw, .3fr, .erf, .mef, .mrw, .pef, .sr2, .srf, .x3f, .fff, .red, .iiq, .srw, .sti, .r3d, .ari, .qtk, .rdc, .hdr, .ia

There are three methods to work around this so that MediaRich can read raw camera files with other extensions:

1. Expand the list of extensions associated with raw camera files.

   If you define a property named *CameraRaw.extensions* in the *local.properties* file, MediaRich will add the extensions listed in the value of this property to the list of extensions associated with raw camera files. The value of this property should be a comma-separated list of extensions that include the . character, just like the list. For example:

   ```
   CameraRaw=.fis,.bla,.abc
   ```

2. Specify `type @ “CameraRaw”` in the load call to indicate that a file should be interpreted as a raw camera file, such as the following:

   ```
   media.load(name @ “image.xyz”, type @ “CameraRaw”);
   ```

3. Specify `detect @ true` in the load call to have MediaRich use the contents of the file to try to figure out how to load it, such as the following:

   ```
   media.load(name @ “image.xyz”, detect @ true);
   ```

Memory Issues with Very Large Image Files

If you are working with very large image files and are experiencing errors or crashes on a Windows system, this could be caused by a low-memory situation. For example, you could find that executing a save() for a 50,000 x 40,000 pixel image after a colorize() with selection() and other operations results in a write error. Such a process requires 8,000,000,000 bytes of memory, and this example would require three rasters of that size to be in memory at the same time.To accommodate this, the system would need 24,000,000,000 bytes, in addition to the operational memory requirements of MediaRich and Windows.

Important: There is NO indication of a low-memory situation caused by something that was done BEFORE the `save()`.

- Check that the automatic scale-down settings are enabled in the Global Properties of the MediaRich CORE Server.

  With the default settings enabled, MediaRich downsizes TIFF, JPEG, and Photoshop files to a max of 8192x8192 at loading. It is possible that this was disabled.

  For more information about setting MediaLoadMaxHeight and MediaLoadMaxWidth to manage automatic scale-down, see the *MediaRich CORE Server Installation and Administration Guide*.

- Try adding more memory to your server.

  For working with large image files, 32 GB of physical RAM is a minimum recommended configuration on Windows systems.

APPENDIX C

File Format Support

MediaRich supports over 400 image, Office, Drawing, and RAW camera file types.

For a regularly updated file format list, please visit our Supported Formats page on the Equilibrium web site:
http://equilibrium.com/mediarichserver/features/#mr-filetypes
http://equilibrium.com/mediarichsharepoint/features/#formats
http://equilibrium.com/equilibriumnew/oneviewer/supported-formats

Appendix summary

MediaRich Image File Formats Natively Supported

The MediaRich CORE provides support for the following image file formats.

Note: If not indicated by a "yes", Text Extraction may or may not work.

File extension and format	Category	Read/Write	Color space support	Metadata ingest	Metadata embed	Text extraction (all text)
PDF: Adobe Acrobat	Portable Document	Read/Write (PDF Image)	Renders to RGB, CMYK	Basic		yes (also one page, page range)
AI: Adobe Acrobat	Drawing-Vector	Read	Renders to RGB, CMYK	Basic		
PIX: Alias Workstation Image	Image	Read/Write	RGB	Basic	None	
RAW: Panasonic Camera Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
CR2: Canon Camera Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
CRW: Canon Camera Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
DNG: Adobe Digital Negative	Digital Negative	Read	Converts to RGB	Enhanced		
EPS: Encapsulated Postscript	Drawing-Vector	Read/Write	Renders to RGB, CMYK	Basic		
EPSF: Encapsulated Postscript	Drawing-Vector	Read/Write	RGB and CMYK	Basic		
RAF: Fuji FinePix Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		

File extension and format	Category	Read/Write	Color space support	Metadata ingest	Metadata embed	Text extraction (all text)
GIF: Graphics Interchange Format	Image/Animation	Read/Write	RGB	Basic	Basic	
JPE: JPEG	Image	Read	RGB, CMYK, with ICC Support	Enhanced		
JPEG: JPEG	Image	Read	RGB, CMYK, with ICC Support	Enhanced		
JPG: JPEG	Image	Read/Write	RGB, CMYK, with ICC Support	Enhanced	Enhanced	
J2K: JPEG 2000	Image	Read	GRAY, RGB, YCBRC Convert	Basic		
JP2: JPEG 2000	Image	Read/Write	GRAY, RGB, YCBRC Convert	Basic	None	
JPX: JPEG 2000	Image	Read	GRAY, RGB, YCBRC Convert	Basic		
J2C: JPEG 2000 Stream	Image	Read	GRAY, RGB, YCBRC Convert	Basic		
JCC: JPEG 2000 Stream	Image	Read	GRAY, RGB, YCBRC Convert	Basic		

File extension and format	Category	Read/Write	Color space support	Metadata ingest	Metadata embed	Text extraction (all text)
JPC: JPEG 2000 Stream	Image	Read/Write	GRAY, RGB, YCBRC Convert	Basic	None	
DCR: Kodak Digital Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
PCT: Mac PICT	Image	Read/Write	RGB	Basic	None	
PICT: Mac PICT	Image	Read	RGB	Basic	None	
MOS: Creo Leaf Mosaic Camera Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
NEF: Nikon Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
ORF: Olympus Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
PCX: PC Paintbrush	Image	Read/Write	RGB	Basic	None	
PS: Photoshop	Image	Read/Write	RGB, CMYK with ICC Support	Enhanced	Enhanced	
PSB: Photoshop Large Image Format	Image	Read	RGB	Enhanced		
PNG: PNG	Image	Read/Write	RGB	Enhanced	Basic	
PBM: Portable Bitmap	Image	Read	B&W	Basic		
PPM: Portable Pixel Map	Image	Read/Write	RGB	Basic	Basic	
PPMB: Portable Pixel Map	Image	Read	RGB	Basic		
PBMA: Portable Pixel Map	Image	Read	B&W	Basic		

File extension and format	Category	Read/Write	Color space support	Metadata ingest	Metadata embed	Text extraction (all text)
PBMB: Portable Pixel Map	Image	Read	B&W	Basic		
PS: Postscript	Portable Document	Read	Renders to RGB, CMYK	Basic		Yes (also one page, page range)
BW: Silicon Graphics Image	Image	Read	RGB	Basic		
RGBA: Silicon Graphics Image	Image	Read	RGB	Basic		
SGI: Silicon Graphics Image	Image	Read/Write	RGB	Basic		
CS1: Sinar Camera Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
ARW: Sony Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
TGA: Targa	Image	Read/Write	RGB	Basic	Basic	
TIF: TIFF	Image/Multi-page image	Read/Write	RGB, CMYK with ICC support	Enhanced	Enhanced	
TIFF: TIFF	Image/Multi-page image	Read	RGB, CMYK with ICC support	Enhanced		
BMP: Windows Bitmap	Image	Read/Write	RGB	Basic	Basic	
WBMP: Wireless Bitmap	Image	Read/Write	RGB	Basic	None	
3FR: Hasselblad Camera Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
ERF: Epson Camera RAW	RAW Camera Image	Read	Converts to RGB	Enhanced		

File extension and format	Category	Read/Write	Color space support	Metadata ingest	Metadata embed	Text extraction (all text)
FFF: Hasselblad Camera RAW	RAW Camera Image	Read	Converts to RGB	Enhanced		
KDC: Kodak Camera Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
MEF: Mamiya Camera Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
MRW: Minolta Camera Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
PEF: Pentax Camera Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
SR2: Sony Camera Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
STI: Sinar Capture Shop Camera Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		
X3F: Sigma Camera Raw	RAW Camera Image	Read	Converts to RGB	Enhanced		

MediaRich Audio and Video File Formats Natively Supported

AVCore 2 includes enhanced performance and GPU transcoding support. It reads all existing formats (EXCEPT FLI, FLC OR ANY VIDEO CONTAINING PALETTED DATA) and writes several existing formats, plus native Flash Video, Ogg Theora, GXF, MTS, and VPX/webm.

MediaRich CORE supports the following audio and video file formats. The video and video/audio formats support 24 bit RGB.

Important: Currently, AVCore 2 is only capable of reading non-paletted video frames.

Audio/Video File Extension and Format	Category	Read/Write	Metadata ingest	Metadata embed
AAC: AAC Audio	audio	Read	Enhanced	None
AC3: AC3 Audio Stream	audio	Read	Enhanced	None
AMR: AMR Audio	audio	Read	Enhanced	None

Audio/Video File Extension and Format	Category	Read/Write	Metadata ingest	Metadata embed
AIF: Audio Interchange File Format (Mac and SGI)	audio	Read, Write	Enhanced	None
AIFF: Audio Interchange File Format (Mac and SGI)	audio	Read, Write	Enhanced	None
DV: Digital Video Stream	video	Read, Write	Enhanced	None
FLV: Flash Video	video	Read, Write	Enhanced	None
F4V: Flash Video	video	Read, Write	Enhanced	None
FLC: Autodesk FLIC Animation File	animation	Read, Write	Enhanced	None
3G: iPhone cell, Mobile 3G	video	Read, Write	Enhanced	None
M4V: iPod, iPhone wifi, Apple TV	video	Read, Write	Enhanced	Enhanced (Transcode only)
MXF: Material Exchange Format	video	Read, Write	Basic	None
3GP: Mobile 3G Audio/Video	video	Read, Write	Enhanced	None
3G2: Mobile 3G Audio/Video	video	Read, Write	Enhanced	None
3GPP: Mobile 3G Audio/Video	video	Read, Write	Enhanced	None
3GP2: Mobile 3G Audio/Video	video	Read, Write	Enhanced	None
MJPG: Motion Jpeg	video	Read, Write	Enhanced	None
MP2: Mpeg Audio Layer 2	audio	Read	Enhanced	None
MPEG: Mpeg1 or 2 Video Program Stream	video	Read, Write	Enhanced	None
MPG: Mpeg1 or 2 Video Program Stream	video	Read, Write	Enhanced	None
M1V: Mpeg1 Video	video	Read, Write	Enhanced	None
MP3: Mpeg2 audio Layer 3	audio	Read	Enhanced	None
VOB: Mpeg2 DVD (unencrypted file required)	video	Read	Enhanced	None
PS: Mpeg2 Program Stream	video	Read, Write	Enhanced	None
MPE: Mpeg2 Standard	video	Read, Write	Enhanced	None

Audio/Video File Extension and Format	Category	Read/Write	Metadata ingest	Metadata embed
MPEG2: Mpeg2 Standard	video	Read, Write	Enhanced	None
MPG2: Mpeg2 Standard	video	Read, Write	Enhanced	None
MP2: Mpeg2 Standard, RAM and RM	video	Read, Write	Enhanced	None
M2T: Mpeg2 Transport Stream	video	Read, Write	Enhanced	None
M2V: Mpeg2 Video	video	Read, Write	Enhanced	None
M4A: Mpeg4 AAC Audio	audio	Read, Write	Enhanced	None
Mpeg4 (RAW): Mpeg2 RAW Video	video	Read	Enhanced	None
MP4: Mpeg4 with AAC and other standard Audio encode	video	Read, Write	Enhanced	None
OGG: OGG Vorbis Audio	audio	Read	Enhanced	None
MOV: Quicktime, Hinted movie * See Codec supported detail	video	Read, Write	Enhanced	Enhanced
RAM: Real Media (writes only 1.0/2.0)	video	Read, Write	Basic	None
RM: Real Media (writes only 1.0/2.0)	video	Read, Write	Basic	None
GXF: Sony Mpeg2-XDCAM SD/HD	video	Read, Write	Basic	None
AU: Sun Microsystems Audio File	audio	Read, Write	Basic	None
AVI: Windows Audio Video Interface	video	Read, Write	Enhanced	None
ASF: Windows Media Active Server (2) (3)	audio/video	Read, Write	Enhanced	Enhanced
WMA: Windows Media Audio (3)	audio	Read, Write	Enhanced	Enhanced
WMV: Windows Media Video (1) (3)	video	Read, Write	Enhanced	Enhanced
Wave: Windows PCM audio format	audio	Read, Write	Enhanced	None

(1) Windows Media 11 for WMV in HD (64-bit Windows Server 2003 not supported)

(2) Sharp Audio Codec must be installed.

(3) When exporting, CBR or Quality-based VBR output settings for audio are the only types supported. Bitrate VBR for audio is not supported.

Basic Metadata support means that MediaRich can extract size, bit depth, frames, and time.

Enhanced Metadata support means there is complete extraction and embedding of metadata with field handlers shown in the documentation. It can embed some kind of information other than width/height/depth/time/samplerate. The types of metadata vary from file format to file format: for instance, not every file format can embed Exif data, or author, etc.

Codecs we have tested and support with legacy QuickTime: Animation, Apple Intermediate Codec, BMP, Cinepak, Component Video, DV - PAL, DV- NTSC, Graphics, H.261, H.263, H.264, JPEG 2000, Motion JPEG A, Motion JPEG B, MPEG-4 Video, Photo - JPEG, Planar RGB, PNG, Sorenson Video, Sorenson Video 3, TGA, and TIFF.

Audio Codecs: 24-bit Integer, 32-bit Floating Point, 32-bit Integer, 64-bit Floating Point, A-Law 2:1, AMR Narrowband, Apple Lossless, IMA 4:1, MACE 3:1, MACE 6:1, MPEG-4 Audio (AAC), Qualcomm PureVoice.

Office File Formats Supported

MediaRich supports the following file formats. They are rendered to RGB. These are read-only on Macintosh and Windows.

Note: If not indicated by a “yes”, Text Extraction may or may not work.

File extension and format	Category	Text extraction (Windows only)
PDB: AportisDoc (Palm)	Word Processing	yes
DXF: AutoCAD Interchange Format	Drawing-Vector	
CGM: Computer Graphics Metafile	Presentation	
XML: DocBook, Microsoft Excel/Word 2003 XML	Word Processing/Spreadsheet	yes
EMF: Enhanced Metafile	Drawing-Vector	
HWP: Hangul WP 97	Word Processing	yes
HTM, HTML: HTML Document	HTML	yes
OTH: HTML Document Template	HTML	yes
PCD: Kodak Photo CD (192x128, 768x512, 384x256)	Drawing-Raster	
123: Lotus 1-2-3	Spreadsheet	yes
WK1: Lotus 1-2-3	Spreadsheet	yes
WKS: Lotus 1-2-3	Spreadsheet	yes
XLSB: Microsoft Excel 2007 Binary	Spreadsheet	yes
XLSM: Microsoft Excel 2007 XML	Spreadsheet	yes

File extension and format	Category	Text extraction (Windows only)
XLSX: Microsoft Excel 2007 XML	Spreadsheet	yes
XLTM: Microsoft Excel 2007 XML Template	Spreadsheet	yes
XLTX: Microsoft Excel 2007 XML Template	Spreadsheet	yes
XLM: Microsoft Excel 4.x-5.0/95/97/2000/XP	Spreadsheet	yes
XLC: Microsoft Excel 4.x-5.0/95/97/2000/XP	Chart	yes
XLW: Microsoft Excel 4.x-5.0/95/97/2000/XP	Spreadsheet	yes
XLT: Microsoft Excel 4.x-5.0/95/97/2000/XP	Spreadsheet	yes
XLS: Microsoft Excel 4.x-5.0/95/97/2000/XP	Spreadsheet	yes
PPTM: Microsoft PowerPoint 2007 XML	Presentation	
PPTX: Microsoft PowerPoint 2007 XML	Presentation	
POTM: Microsoft PowerPoint 2007 XML Template	Presentation	
POTX: Microsoft PowerPoint 2007 XML Template	Presentation	
PPS/PPSX: Microsoft PowerPoint 97/2000/XP	Presentation	
PPT: Microsoft PowerPoint 97/2000/XP	Presentation	yes
POT: Microsoft PowerPoint 97/2000/XP Template	Presentation	
DOC: Microsoft WinWord, 5, 6.0/95, 97/2000/XP	Word Procession	yes
DOCM: Microsoft Word 2007 XML	Word Procession	
DOCX: Microsoft Word 2007 XML	Word Procession	yes
DOTM: Microsoft Word 2007 XML Template	Word Procession	yes
DOTX: Microsoft Word 2007 XML Template	Word Procession	yes
DOT: Microsoft Word 95, 97/2000/XP Template	Word Procession	yes
OTG: ODF Drawing Template	Drawing-Vector	
ODG: ODG Drawing, ODF (Impress)	Drawing-Vector	
ODM: ODF Master Document	Word Processing	yes
ODP: ODF Presentation	Presentation	
OTP: ODF Presentation Template	Presentation	
ODS: ODF Spreadsheet	Spreadsheet	yes

File extension and format	Category	Text extraction (Windows only)
OTS: ODF Spreadsheet Template	Spreadsheet	yes
ODT: ODF Text Document	Word Processing	yes
OTT: ODF Text Document Template	Word Processing	yes
STD: OpenOffice 1.0 Drawing Template	Drawing-Vector	
SXD: OpenOffice 1.0 Drawing, OpenOffice Impress	Drawing-Vector	
STW: OpenOffice HTML/Text Template	HTML	
SXG: OpenOffice 1.0 Master Document	Word Processing	yes
SXI: OpenOffice 1.0 Presentation	Presentation	
SXC: OpenOffice 1.0 Spreadsheet	Spreadsheet	yes
STC: OpenOffice 1.0 Spreadsheet Template	Spreadsheet	yes
SXW: OpenOffice 1.0 Text Document	Word Processing	yes
MET: OS/2 Metafile	Drawing-Vector	
PXL: Pocket Excel	Spreadsheet	yes
PSW: Pocket Word	Word Processing	yes
PGM: Portable Graymap	Drawing-Raster	
WB2: Quattro Pro 6.0	Spreadsheet	yes
RTF: Rich Text Format, RTF (OpenOffice Calc)	Word Processing	yes
SVG: Scalable Vector Graphic	Drawing-Vector	
SGV: StarDraw 2.0	Drawing-Vector	
SDP: StarImpress 4.0/5.0	Presentation	
SVM: StarView Metafile	Drawing-Vector	
SDW: StarWriter 1.0, 2.0, 3.0-5.0	Word Processing	yes
SGL: StarWriter 4.0/5.0 Master Document	Word Processing	yes
SGF: StarWriter Graphics Format	Drawing-Vector	
STW: StarWriter doc	Word Processing	
SXD: OpenOffice 1.0 drawing	Drawing	
STD: OpenOffice 1.0 drawing	Drawing	

File extension and format	Category	Text extraction (Windows only)
RAS: Sun Raster Format	Drawing-Raster	
SLK: SYLK	Spreadsheet	yes
602: T602 Document	Word Processing	yes
UOS: Unified Office Format spreadsheet	Spreadsheet	yes
UOF: Unifed Office Format spreadsheet, text, presentation	Spreadsheet	yes
UOP: Unified Office Format presentation	Presentation	
WMF: Windows Metafile	Drawing-Vector	
WPD: WordPerfect Document	Word Processing	yes
XBM: X Bitmap	Drawing-Raster	
XPM: X PixMap	Drawing-Raster	

Index