BrightSign®

DEVELOPER'S GUIDE

BrightSign Media Server
BrightSign Firmware Version: 6.0.x
BrightSign Models: 4K242, 4K1042, 4K1142, XD232, XD1032, XD1132

BrightSign, LLC. 16780 Lark Ave., Suite B Los Gatos, CA 95032 | 408-852-9263 | www.brightsign.biz

TABLE OF CONTENTS

INTRODUCTION

This Developer's Guide applies to 4Kx42 and XDx32 players running firmware versions 6.0.x or later. Most BrightSign models can stream files on local storage to other sources on the local network, but the 4Kx42 and XDx32 are the only models that support encoding and transcoding of multimedia sources.

This document first outlines how to perform unicast and multicast streaming using the *roMediaStreamer* object. It then outlines how to use the *roMediaServer* object to deploy a 4Kx42 and XDx32 player as a media server. The final section discusses performance estimates and limitations of the Media Server.

Currently, all server applications are handled using BrightScript. See the BrightScript Reference Manual and BrightScript Object Reference Manual for more details about developing with BrightScript.

Note: *4Kx42 models can stream 4K (H.265) content from a file or an incoming stream, but they cannot encode or transcode 4K content.*

MEDIA STREAMER

The media streamer is controlled by setting a pipeline configuration. The stream-source component and stream-destination component are specified by passing a string to the *roMediaStreamer.SetPipeline()* method. This string contains both the source and destination components concatenated together; the different stages of the pipeline are delimited by commas.

The pipeline is started by calling the `start()` method (without extra arguments). The `stop()` method can be used to stop the pipeline. However, some of the pipeline stages (described in the Media Streamer State Machine section below) will continue running internally, so the `reset()` method may be preferable for terminating the stream.

Simple File Streaming

The following example shows how to stream from a file source to a single client:

```
m = CreateObject("roMediaStreamer")
m.SetPipeline("file:///data/clip.ts, udp://239.192.0.0:1234/")
m.Start()
```

To stop the stream, call `m.reset()`.

The `file` source and `udp`/`rtp` destinations do not function the same as in 4.7 firmware. However, the 4.7 method of file streaming is still available using the `filesimple`, `udpsimple` and `rtpsimple` components. The following example works exactly the same as media streaming in 4.7:

```
m.SetPipeline("filesimple:///data/clip.ts, udpsimple://239.192.0.0:1234/")
```

Note that the `filesimple` component only works with `udpsimple`/`rtpsimple`, and vice versa.

Media Streamer State Machine

The media streamer, represented by the `SetPipeline()` method, has four states:

- RESET: The media streamer has been created, but nothing is allocated or running.
- INITIALISED: Some allocation may have happened and some resources may be reserved, but nothing is running.
- CONNECTED: All structures have been created and all the pipeline components are connected together. Though the pipeline is producing no output, some internal parts of it may already be running.
- RUNNING: The pipeline is running.

When a media streamer is created, it starts in the RESET state. After calling `SetPipeline()`, the states can be progressed through using the following *roMediaStreamer* methods: `Initialise()`, `Connect()`, and `Start()`. Moving to a later state causes any intervening states to be traversed automatically, which is why a pipeline can be started by simply calling `Start()`.

When a media streamer is in the RUNNING state, it can be moved backwards through the states by calling, respectively, `Stop()`, `Disconnect()`, and `Reset()`. As before, intermediate states can be omitted, so it is possible to stop the media streamer and de-allocate all its resources simply by calling `Reset()`. While `Stop()` does stop any output from emerging from the pipeline, there may still be some internal activity.

Calling `SetPipeline()` always causes the media streamer to return to the RESET state. All of the above interface functions, aside from `SetPipeline()`, take no arguments.

Source Components

This section lists the currently available source components for a streaming pipeline. Note that some source components (i.e. `mem:/` and `memsimple:/`) cannot be called until they are first created using destination components, which are described in the next section.

filesimple:///filename

This component reads the named file from the local storage. The file must be an MPEG-2 transport stream (usually with a *.ts* suffix). This component can only be connected to `udpsimple:`, `rtpsimple:`, or `memsimple:` components. The optional `loop` parameter can be appended to cause the input file to loop:

```
m.SetPipeline("filesimple:///file.ts?loop, udpsimple://239.192.0.0:1234/")
```

file:///filename

This component can act as either a source or a destination. As a source, it reads from an MPEG-2 transport stream file and can be connected more generally to other components. This non-simple component must be used when connected to `hls:` destination components (because it uses an indexed stream). Because it uses fewer resources, we recommend using the `filesimple:` component instead of the `file:` component whenever it is sufficient.

This component accepts the optional `key` and `iv` parameters for file encryption/decryption.

The optional `loop` parameter can be appended to cause the input file to loop:

```
m.SetPipeline("file:///file.ts?loop, hls:///file?duration=3")
```

hdmi:[///]

This component receives audio and video from the HDMI Input. It can then be fed through an encoder component and streamed or written to a file.

display:[///]

This component receives the audio and video of the current presentation, allowing the player to encode and stream its current display output. See the Display Encoding chapter for more information about using this component.

udp://IP_address:port/

This component receives a stream using the UDP protocol.

rtp://IP_address:port/

This component receives a stream using the RTP protocol.

rtsp://IP_address:port/path

This component receives a stream using the RTSP protocol. An optional `dtcp.port` parameter can be included to specify a DTCP-IP encrypted session:

```
rtsp://10.1.243:554/stream1?dtcp.port=8888
```

http://IP_address:port/path

This component receives a stream using the HTTP protocol.

https://IP_address:port/path

This component receives a stream using the HTTPS protocol.

gst://IP_address:port/path

This component receives a GStreamer pipeline and generates a non-simple stream using it.

gstsimple://IP_address:port/path

This component receives a GStreamer pipeline and generates a simple stream using it.

memsimple:/name/stream.ts

This component reads from a previously created (destination) memory stream component with the given `name`, and forwards the results to other simple components. Note that `/stream.ts` is appended to denote "the entire memory buffer". The following example will multicast a memory stream:

```
m.SetPipeline("memsimple:/livestream/stream.ts, rtpsimple://239.192.0.0:5004/")
```

A `memsimple:` source component can originate from either a simple memory stream (using the `memsimple:` destination component) or a non-simple, "indexed" memory stream (using the `mem:` destination component).

See the Memory Streaming section for more details.

mem:/name/stream.ts

This component reads from a previously created destination memory stream with the given `name`; it can then forward the results to other non-simple components. Note that `/stream.ts` is appended to denote "the entire memory buffer". Assuming sufficient resources, the following example will transcode and stream a memory stream:

```
m.SetPipeline("mem:/livestream/stream.ts, decoder:, encoder:, rtp://239.192.0.0:5004/")
```

General Source Component Options

All source components can accept the `encryption` parameter, which instructs the pipeline to behave as though the source is copy-protected, whether or not it actually is. This is helpful in forcing the destination components to be initialized with copy-protection even if the source is initially unprotected. Currently, DTCP encryption is the only available encryption method:

```
"file:///file.ts?encryption=dtcp"
```

Destination Components

This section lists the currently available destination components.

udpsimple://IP_address:port/

This component streams its input over UDP to the given IP address and port. This destination only works with the `filesimple:` or `memsimple:` source component as input.

rtpsimple://IP_address:port/

This component streams its input over RTP to the given IP address and port. This destination only works with the `filesimple:` or `memsimple:` source component as input.

httpsimple:///socket=num

This component streams its input over an HTTP connection. This destination only works with the `filesimple:` or `memsimple:` source component as input. It is intended for use as an HTTP media server and cannot be used to stream directly to a client.

udp://IP_address:port/

This component streams its input over UDP to the given IP address and port.

rtp://IP_address:port/

This component streams its input over RTP to the given IP address and port.

http:///socket=num

This component streams its input over an HTTP connection. This component is intended for use as an HTTP media server and cannot be used to stream directly to a client.

file:///filename

This component writes its input to the named file, which will be saved as an MPEG-2 transport stream file. This component accepts the optional `key` and `iv` parameters for file encryption.

display:[///]

This component allows you to see what's in a stream (often in conjunction with the decoder component) and is provided for debugging only: BrightScript primarily utilizes the *roVideoPlayer* and *roAudioPlayer* objects to play back streaming content.

memsimple:/name

This component works only with the `filesimple:` component. It writes its input to a memory stream named `/name`. This constitutes a simple memory stream, meaning that it is not indexed and cannot be input to a `mem:` source, only another `memsimple:` source. This component takes an optional size parameter that specifies the size of the memory buffer in megabytes. The default memory buffer size is 5MB.

```
m.SetPipeline("filesimple:///file.ts, memsimple:/file?size=2")
```

mem:/name

This component creates a memory stream with the specified `/name`. This component can receive input from general components and writes to an indexed (rather than simple) memory stream, which can be re-read by the `mem:` source component. It can therefore be used to support HLS streaming.

```
m.SetPipeline("file:///file.ts, mem:/file?duration=3&size=25")
```

The `mem:` destination component accepts two optional parameters:

- `size`: The size of the memory buffer in MB (megabytes). For HLS streaming, this needs to be a large buffer.
- `duration`: The target duration of the index points (which become HLS segments).

hls:///path

This component segments the incoming stream and writes it to the given path. The written segments will have the name `path_000000.ts`, `path_0000001.ts`, etc., and the index file will be named `path_index.m3u8`.

The HLS destination component accepts the parameter `duration`, which specifies the approximate target length, in seconds, of the HLS segments. The default value is `5`, so to segment a file into 3 second durations you could use code similar to the following:

```
m.SetPipeline("file:///file.ts, hls:///file?duration=3")
```

The above function would split the `file.ts` file into segments of approximately 3 seconds each, named `file_000000.ts`, `file_000001.ts`, `file_000002`, etc.

Destination Component Options

The following options apply to the `udpsimple`, `rtpsimple`, `udp`, and `rtp` destination components.

maxbitrate

The `maxbitrate` parameter throttles the maximum instantaneous bitrate (in Kbps) of a stream.

Example: The following component would never stream at a rate greater than approximately 2Mbps:

```
rtp://IP_address:port/?maxbitrate=2000
```

This number needs tuning both for the content being streamed and the network over which they are being sent. Note that it can be challenging to configure bitrates for some wireless networks.

ttl

The `ttl` parameter specifies how many times multicast packets can be forwarded across switches and other network infrastructure. The default value for this parameter is `1`.

Example:

```
rtp://IP_address:port/?ttl=64
```

Other Components

encoder:[///][param=value]

This component receives an un-encoded input and outputs audio/video data encoded as an MPEG-2 transport stream. The following are valid parameters:

- `audiodelay=[delay_in_ms]`: The audio synchronization offset in milliseconds. The default value is 0. A positive value will delay the audio with respect to the video, while a negative value will delay the video with respect to the audio.
- `vbitrate=[bitrate]`: The video bitrate specified in Kbps.
- `vformat=[format]`: The resolution and frame rate of the video output. The `format` can be specified as any of the following:

- `480i50`
- `480p25`
- `720p24`
- `720p25`
- `720p30`
- `720p50`
- `720p60`
- `1080i50`
- `1080i60`
- `1080p24`
- `1080p25`
- `1080p30`
- `1080p50`
- `1080p60`

This component utilizes H.264 for video and AAC for audio. The default video format is 720p30, and the default bitrate is 6Mbps.

The following example encodes video from the HDMI Input and writes it to the local storage as a *.ts* file:

```
m.SetPipeline("hdmi:, encoder:vformat=480p25&vbitrate=1000, file:///hdmi.ts")
```

esencoder:[///][param=value]

This component receives an un-encoded input and outputs video data as an elementary H.264 stream (as opposed to the standard `encoder:` component, which outputs an MPEG-2 transport stream). The H.264 stream has no audio data. This component accepts the same `vformat` and `vbitrate` parameters as the `encoder:` component.

Since non-simple components expect an MPEG-2 transport stream, this component only works with the following: `rtpsimple:`, `udpsimple:`, `httpsimple:`, and `memsimple:`.

decoder:[///]

This component receives an encoded input and decodes it. The primary uses of this component are to pass a video stream to the `display:` destination for playback or to pass a video file to an `encoder:` component for transcoding.

HDCP and DTCP

The *roMediaStreamer* object will honor the copy-protection status of its inputs by assigning the same status to its outputs. Each stream can have one of three levels of copy protection:

- **None**: No copy protection is enabled. These streams can be saved to files or streamed over the network without restriction.
- **DTCP**: The input is protected with DTCP. A DTCP-protected stream cannot be saved to a file, but it can be streamed over the network if the IP-streaming component has successfully authenticated against a DTCP-enabled client.
- **HDCP**: The input is protected with HDCP. It can be neither saved nor streamed, but it can be output to an HDCP-authenticated display.

HDCP Workflow

The *roMediaStreamer* object will handle HDCP authentication and de-authentication as follows:

- If a non-HDCP authenticated stream becomes HDCP authenticated, any file or IP-streaming destination components will stop immediately and send an `EOS_ERROR` event. If the connected display cannot be HDCP authenticated, the display destination will hide the video output .
- If a stream that was previously HDCP authenticated becomes unprotected, the file and IP streaming destination components *will not* be resumed. However, if the display destination is not HDCP authenticated, it will begin displaying video. An HDCP-authenticated display will be unaffected either way.

Encryption Flags

The encryption level can be forced by flagging it on the source component. For example, the following source component will force any connected display receiving HDMI input to be HDCP authenticated:

```
hdmi:encryption=hdcp
```

The following source component would force a DTCP-IP negotiation to occur before the stream is output to the network:

```
file:///clip.ts?encryption=dtcp
```

Note: *DTCP-IP requires a two-way negotiation and is, therefore, only supported when it is initiated via the RTSP protocol.*

Important: *The HDCP or DTCP encryption status of a stream cannot be removed by setting the `encryption` parameter to `none`.*

File Encryption

The `file:///` source and destination components accept the optional `key` and `iv` parameters, which can be used to encrypt or decrypt a file using the AES CTR algorithm. The `key` and `iv` values must be specified as 16-byte hex strings.

Note that this file encryption process is distinct from HDCP and DTCP authentication. There are no restrictions on how encrypted files may be used in a pipeline, but you cannot use this form of encryption on a stream that was originally HDCP or DTCP-IP protected.

The following example saves an HLS stream as an encrypted file. Once the file is written, it can be decrypted from a `file:///` source component using the same `key` and `iv` values:

```
m = CreateObject("roMediaStreamer")
m.SetPipeline("http://<ip-address>/playlist.m3u8,file:///file.ts?key=
30313233343536373839616263646566&iv=30313233343536373839616263646566")
```

```
m.Start()
```

Streaming Examples

The following code snippets provide examples for common types of media streaming.

Streaming an MPEG-2 TS File over UDP

To initialize the *roMediaStreamer* instance and begin the stream, use the following:

```
m = CreateObject("roMediaStreamer")
m.SetPipeline("file:///data/clip.ts, udp://239.192.0.0:1234/")
m.Start()
```

To stop the stream, use the following:

```
m.reset()
```

You can also use the simple form of components for most common streaming tasks. The following would operate identically to the above stream, while using fewer resources:

```
m.SetPipeline("filesimple:///data/clip.ts, udpsimple://239.192.0.0:1234/")
```

Streaming an Encoded HDMI Input over RTP

The resolution and bitrate of the streamed video can be changed by adding parameters to the `encoder:` component. Use the following code to stream the input using the native resolution and default bitrate:

```
m = CreateObject("roMediaStreamer")
m.SetPipeline("hdmi:, encoder:, rtp://239.192.0.0:1234/")
m.Start()
```

To stop the stream, use the following:

```
m.reset()
```

Recording a File

The following code will record an HDMI Input and save it as a file named *hdmi.ts*:

```
m = CreateObject("roMediaStreamer")
m.SetPipeline("hdmi:, encoder:, file:///hdmi.ts")
m.Start()
```

To stop the recording process, use the following:

```
m.reset()
```

Transcoding and Streaming a File

The output of a `decoder:` component can be connected to the input of an `encoder:` component to transcode the data. This is useful if you want to modify the bitrate of the video before streaming it over the network. The following code will transcode the *file.ts* video before streaming it over RTP:

```
m = CreateObject("roMediaStreamer")
m.SetPipeline("file:///file.ts, decoder:, encoder:vbitrate=1000, rtp://239.192.0.0:5004/")
m.Start()
```

To stop the transcoding process, use the following:

```
m.reset()
```

Re-streaming

The player can be used to output a streaming input using a different protocol by connecting an IP client component to an IP server component. The following code will receive an HTTP stream and then stream it using RTP:

```
m = CreateObject("roMediaStreamer")
```

```
m.SetPipeline("http://172.30.1.37/file.ts, rtp://239.192.0.0:5004/")
m.Start()
```

To stop the stream, use the following:

```
m.reset()
```

Segmenting a File Using HLS

An existing *.ts* file can be segmented for streaming using the HLS protocol. The resulting file segments can then be streamed directly by making HTTP requests to an HTTP Media Server. The following code will use the *file.ts* video to generate segments that are approximately 10 seconds in duration:

```
m = CreateObject("roMediaStreamer")
m.SetPipeline("file:///file.ts, hls:///media_segment?duration=10")
m.Start()
```

In this case, the files will begin with *media_segment_000000.ts*, and the counter will increment once for each segment. The index file will be written as `media_segment_index.m3u8`.

Simple and Non-Simple Components

To conserve system resources, we recommend using the simple versions of components whenever possible. Use the following guidelines to determine whether simple or non-simple components should be used:

- Use *simple components* when streaming a file from local storage using UDP, RTP, or HTTP protocols (without any encoding or transcoding involved in the pipeline).
- Use *non-simple components* whenever the pipeline includes encoding, transcoding, or HLS streaming.
- When ingesting a stream from a remote source and re-streaming it, use the `udpsimple:`, `rtpsimple:`, and `httpsimple:` destination components. If you need to modulate the stream (or re-stream using HLS), use *non-simple components*.

- Since a `memsimple:` source component can read from a `mem:` destination component, use the `memsimple:` source component in conjunction with `udpsimple:`, `rtpsimple:`, or `httpsimple:` when memory streaming. The `mem:` source component should only be used with the `hls:` destination component (or `file:` if debugging).

MEMORY STREAMING

As outlined above, the *roMediaStreamer* object provides a means of specifying a single flexible pipeline of multimedia processing. However, the pipeline is constrained to having a single input and a single output. The *memory streaming* feature adds a means of bifurcating the data stream so that multiple outputs can be created, and more than one thread of processing can be realized.

Note that memory streaming is especially important when serving live-encoded media such as HDMI Input. Memory streaming ensures that only a single encoder is needed to distribute a live-encoded stream to multiple clients.

HTTP Multiple Unicasting

To unicast a source stream to multiple clients, output the source (in this case, encoded HDMI Input) to a memory stream:

```
m = CreateObject("roMediaStreamer")
m.SetPipeline("hdmi:, encoder:, mem:/livehdmi")
m.Start()
```

Now a large number of clients can receive the same stream named "livehdmi" (the name is arbitrary but must not conflict with any other memory stream).

When a client accesses the following URL, the HTTP media-server code creates a separate *roMediaServer* instance reading from the source `mem:/livehdmi/stream.ts` and forwards it to the HTTP socket:

```
http://IP_address:port/mem:/livehdmi/stream.ts
```

Notice that `/stream.ts` must be appended to the memory stream name to denote that the entire stream should be used.

Note: *In the above instance, the media streamer code will actually create the source component* `memsimple:/livehdmi/stream.ts` *as it uses fewer resources and is more efficient.*

Once the memory stream has been created, it can also be simultaneously multicast using another Media Streamer instance:

```
m1 = CreateObject("roMediaStreamer")
m1.SetPipeline("memsimple:/livehdmi/stream.ts,rtpsimple://239.192.0.0:5004/")
m1.Start()
```

mem: and memsimple:

As with certain other components, there is a "simple" and a "non-simple" version of memory streaming. The simple version (`memsimple:`) can only be used in a pipeline with other simple components, although simple memory streams can read from non-simple (`mem:`) memory streams (as shown above, where the `memsimple:/livehdmi` component reads from the `mem:/livehdmi` component). However, the converse is not possible: The `mem:` source component cannot be used to read from a memory stream that was created with the `memsimple:` destination component.

Because simple memory streams are not indexed, they cannot support HLS streaming operations downstream (other components will not receive information about the HLS segments).

As is the case with other "simple" components, simple memory streams should be used wherever possible because they consume fewer resources.

HLS Multiple Unicasting

HLS multiple unicasting functions similarly to the HTTP multicast example given above, but with one exception: Stream indexing filters only work on streams read into the transport stream processor, not on streams read into the encoder (both of which are handled by the `mem:` component). As a result, encoded streams such as HDMI Input must be passed to the `mem:` component twice before they can be multicast. This is easily accomplished using two `SetPipeline()` calls:

```
m = CreateObject("roMediaStreamer")
m1 = CreateObject("roMediaStreamer")
```

```
m.SetPipeline("hdmi:,encoder:,mem:/hdmi_unindexed?size=5")
m1.SetPipeline("mem:/hdmi_unindexed/stream.ts ,mem:/hdmi_indexed?size=30&duration=5")
m.Start()
m1.Start()
```

Note that a larger memory buffer (30MB in this instance) must be specified for the second `mem:` component because it must hold at least five complete 5-second segments at any time. Clients generally require 5 complete segments at all times for HLS streams, so they will often need to wait at least 25 seconds before the stream can be played successfully.

Clients can then access HLS streams using the following URL:

```
http://IP_address:port/mem:/hdmi_indexed/index.m3u8
```

Like HTTP multicasting, the media streamer will generate additional *roMediaStreamer* instances for each client that connects. In addition to HLS clients, HTTP and RTSP clients can also connect to the indexed stream using the following URLs:

HTTP

```
http://IP_address:port/mem:/hdmi_indexed/stream.ts
```

RTSP

```
rtsp://IP_address:port/mem:/hdmi_indexed/stream.ts
```

Note: *The indexed stream can also be multicast by another* roMediaStreamer *instance.*

Accessing an Indexed Memory Stream

As shown in previous examples, appending `stream.ts` to the memory stream identifier denotes the entire buffer, while appending `index.m3u8` to an indexed (HLS) stream denotes the entire index file. Similarly, appending the name of a segment listed in the index file will access that segment for the duration of its lifetime in the memory stream buffer. For example, if the index file lists a segment named `000179.ts` as available in the `mem:/livestream` component, then the following source component can be used to fetch that segment:

```
mem:/livestream/000179.ts
```

Note: *The `memsimple:` component can also be used to fetch the segment.*

Memory Stream Pacing

The flow of data into the memory stream is always paced in real time to give the reading clients time to keep up. An *roMediaStreamer* instance that is writing will never wait for any of the reading clients; if any client gets too far behind (which is detected by the writing edge of the FIFO overtaking the client's reading edge), the client will abort and issue an EVENT_EOS_ERROR message.

Because of this limitation, it is important to note the following: If a memory stream is used to work around the fact that data cannot be passed directly from an encoder to a decoder (as in the below example), then processing cannot happen faster than in real time.

Example

```
m1 = CreateObject("roMediaStreamer")
m2 = CreateObject("roMediaStreamer")
m1.SetPipeline("hdmi:,encoder:,mem:/hdmi")
m2.SetPipeline("mem:/hdmi/stream.ts,decoder:,file:///file.ts")
m1.Start()
m2.Start()
```

Memory Stream States

Note: *See the State Machine section for more information on* roMediaStreamer *states*.

A memory stream is only created when the Media Streamer that names it as a destination is in at least the CONNECTED state. This means it is possible to connect the pipeline (using `Connect()`) that creates the memory stream but not start it. Any number of Media Streamers reading from a memory stream can be created (and even started) before the original Media Streamer is itself started, beginning the flow of data into the memory stream.

Encryption and Copy Protection

Generally speaking, an *roMediaStreamer* instance reading from a memory stream will adopt the same encryption/copy-protection as the associated *roMediaStreamer* instance that is writing to it. The specific rules are as follows:

- **Simple writer**: A `memsimple:` destination cannot be marked as requiring protection (nor can it receive such a stream).
- **Simple reader**: A `memsimple:` source will only read from a memory stream that was marked as unencrypted (requiring no protection) when it was created.
- **Non-simple writer**: A `mem:` destination will be marked as requiring protection when it is created according to the source of the pipeline. If a stream becomes protected later, it will continue to operate if it was marked as requiring protection initially (optionally with the `encrypt` parameter), otherwise it will stop.
- **Non-simple reader**: A `mem:` source inherits the protection status of its associated writer, and therefore its output will ultimately require HDCP or DTCP protection if the source was similarly marked.

DISPLAY ENCODING

The `display:` source component allows the player to accurately stream the graphics and video output of the current presentation (without audio). Once encoded, the display can be used like any other stream: It can be unicast, multicast, saved to a file, passed to a memory stream, or accessed via HTTP or RTSP servers. There are some limitations to what can be streamed, but the `display:` component is able to provide a near facsimile of what is currently being output on the display.

Example

```
m.SetPipeline("display:mode=1,encoder:,rtp://239.192.0.0:5004/")
```

Modes and Limitations

The `display:` component has two video streaming modes, which are specified as follows:

```
display:mode=[mode_number]
```

- **Mode 1**: Like the primary display, the stream supports two video zones. This guarantees that the stream will include all videos in the display area. However, the stream will not scale graphics properly, so it is best not to specify a display resolution: In other words, the resolution of the display source will need to match that of the stream destination.
- **Mode 2**: The graphics can be properly scaled to any size, but the stream can only support one video zone. This means that, depending on the presentation, the stream may not always include all videos in the display area.

Video Transforms

Neither streaming mode supports displaying a video window that has a transformation applied to it (for example, by calling *roVideoPlayer.SetTransform()* in BrightScript). In these cases, the video zone will appear in the encoded display stream, but without the transform applied to it (the primary display will be unaffected either way).

HDCP and DTCP

If a video zone is playing from a source that is HDCP or DTCP protected, that video will not appear in the encoded display stream. The video will still appear on a display connected to the player that is streaming.

Display Resolutions

Similar to the `encoder:` component, the resolution of an encoded display stream can be specified using the `vformat` parameter.

```
display:mode=[mode_number]&vformat=[resolution]
```

As noted above, the video `mode` will usually need to be set to `2` to ensure accurate scaling of graphics in the presentation. For display encoding, the `vformat` parameter of the `encoder:` destination component is ignored. However, the `vbitrate` parameter of the `encoder:` destination component is still used to specify the bitrate of the encoded display stream.

When the `vformat` parameter is not specified, the encoded video stream will match the resolution of the primary display (i.e. the video output of presentation). This means that the video `mode` can be set to `1` without causing graphics-scaling issues. The `vformat` parameter currently accepts the following values:

- `480i` (running at 50Hz)
- `480p25`
- `480p30`
- `720p60`
- `720p50`
- `720p24`
- `720p30`
- `1080i60`
- `1080i50`
- `1080p24`

- `1080p25`
- `1080p30`
- `1080p50`
- `1080p60`

MEDIA SERVER

The *roMediaStreamer* object allows you to configure a pipeline of multimedia processing elements for execution. This may involve streaming the results over RTP or UDP, but it is not capable of behaving as a true server, which listens for connections and then acts upon them. This is the job of the Media Server, represented by the *roMediaServer* object.

The Media Server waits for requests, deals with any negotiation, and ultimately creates a Media Streamer pipeline which it executes to fulfill the request. The Media Server currently supports the RTSP protocol (as used, for example, by VLC), and HTTP requests. These requests from the client must take the following form:

```
protocol://IP_address:port/media_streamer_pipeline
```

- `protocol`: Either `rtsp` or `http`
- `IP_address:port`: The IP address of the BrightSign player and the port number on which the Media Server is running. For more information, refer to the example below.
- `media_streamer_pipeline`: A Media Streamer pipeline as given in previous examples, but without the final destination component (as the destination is implicit in the request from the client).

Initializing the Media Server

An RTSP Media Server can be started as follows:

```
s = CreateObject("roMediaServer")
s.Start("rtsp:port=554")
```

This will start an RTSP server listening on port 554. The port number and streaming protocol can be customized: For example, an HTTP server can be started on port 8080 instead as follows:

```
s = CreateObject("roMediaServer")
s.Start("http:port=8080")
```

The media server supports a number of optional parameters after the `port` parameter, which may be appended to the command string with an "&" (ampersand):

- `port`: Specifies a port number for the server. If this parameter is not specified, the server defaults to 554 for RTSP and 8080 for HTTP.
- `trace`: Displays a trace of messages in the negotiation with the client. This parameter is useful primarily for debugging RTSP sessions. For example: `rtsp:port=554&trace`
- `maxbitrate`: Sets the maximum instantaneous bitrate (in Kbps) of the RTP transfer initiated by RTSP. This parameter has no effect for HTTP. The parameter value 80000 (i.e. 80Mbps) has been found to work well. The default behavior (also achieved by passing `0`) is to not limit the bitrate at all. For example: `rtsp:port=554&trace&maxbitrate=80000`
- `threads`: Sets the maximum number of threads the server is prepared to have running. Each thread handles a single client request. The default value is `5`. For example: `http:port=8080&threads=10`

To stop the Media Server, use the `Stop()` method. This actually signals all the threads to stop, but does not wait for this to happen. To block until everything has truly finished, either use `s.Terminate()` (which may also be used on its own), or simply allow the Media Server object to be subject to garbage collection.

Media Server Examples

These examples are client-side URLs that can be pasted into VLC for testing. Note that spaces between the Media Streamer pipeline components, as well as filenames containing commas, are not permitted.

Requesting a File To Be Streamed

Use the following URL to request the server to stream a file from local stoare.

```
rtsp://IP_address:port/file:///file.ts
```

The `loop` parameter can be appended to loop the file indefinitely:

```
rtsp://IP_address:port/file:///file.ts?loop
```

Use the following to stream the `file.ts` using HTTP instead:

```
http://IP_address:port/file:///file.ts
```

Requesting an Encoded HDMI Input Stream

Use the following URL to request the server stream its HDMI Input:

```
rtsp://IP_address:port/hdmi:,encoder:
```

Requesting a Memory Stream

Memory streams that have been previously configured can be requested either by RTSP or HTTP. The following example will stream the memory stream named `name`, which has been previously set running:

```
http://IP_address:port/mem:/name/stream.ts
```

Note that `/stream.ts` is appended to denote the entire stream, rather than merely portions of it (as in the HLS case).

Requesting an HLS Stream

An indexed (i.e. non-simple) memory stream component must be already running to service an HLS request. HLS streaming can be initiated by requesting the following URL:

```
http://IP_address:port/mem:/name/index.m3u8
```

This will fetch the playlist file for the memory stream named `name`. Observe that `/index.m3u8` must be appended to denote the *index file*, rather than the stream itself. Alternatively, if the HLS index and segment files have been pre-saved onto the storage of the server, they can be accessed by regular HTTP requests.

Media Server and DTCP-IP

Because the Media Server simply receives a *roMediaStreamer* pipeline designation in the URL from the client, it is recommended that the client explicitly specify that it is requesting an encrypted session. To specify encryption, add the `encrypt` parameter to the source component. This instructs the source to behave as if it is copy-protected (whether or not this is true). The `dtcp.port=8888` parameter is actually stripped off on the client side and used to perform the DTCP negotiation with the server on port 8888 (in this example).

DTCP-IP can also be used simply to encrypt content over the air. To stream a file so that it cannot be snooped by others, the client URL may be formatted as follows:

```
rtsp://IP_address:port/file:///file.ts?encrypt?dtcp.port=8888
```

URL Syntax for an RTSP Client

When specifying the URL, a BrightSign player acting as a client may interpret parts of the URL after a "?" (question mark) as options for it to parse; likewise, the stages on the Media Server that fulfill the client request may expect the same. For example, in the following URL, it may be ambiguous whether the `loop` parameter is destined for the player on the client side or the media streamer on the server side:

```
rtsp://IP_address:port/file:///file.ts?loop
```

To clarify this ambiguity, the client-side player will look for parameters after the final "?", though it will ignore parameters that it does not recognize. To ensure that all parameters are going through to the server side, append an additional final "?" to the URL:

```
rtsp://IP_address:port/file:///file.ts?loop?
```

Multi-Device Pipeline Stages

When one player is acting as the streaming client, and another player is acting as the Media Server, it may at times be necessary for the client to define pipeline stages on the Media Server. For example, the client may require the Media

Server to encode its HDMI Input before streaming it over HTTP (because a server cannot stream raw HDMI video frames over the network). In these cases, you can use parentheses in the client request to delineate pipeline stages on the server:

```
(http://IP_address:port/hdmi:,encoder:),file:///hdmi.ts
```

In the above example, the client-side code instructs the Media Server to encode HDMI input and stream it to the client, which then saves it as a file.

Remote Pipelines

It is possible to have a client player specify pipeline stages on the Media Server only. For example, the client can have the Media Server initialize a multicast stream, without actually connecting to the stream. The following code will command the Media Server to initialize a multicast stream using a file on its local storage:

```
(remote://IP_address:port/filesimple:///file.ts,rtpsimple://239.192.0.0:5004/)
```

The `remote:` protocol and encompassing parentheses indicate to the server that this is a fully self-contained pipeline to execute. No media is streamed to the client, but any signals (including end-of-stream and error messages) are forwarded across the socket to the client, which can use the *roMessagePort* object to receive such messages.

Resetting the *roMediaStreamer* instance on the client side will force the socket connection to close, and will thus terminate the pipeline on the server as well.

MEDIA SERVER PERFORMANCE

The number of streams a 4Kx42 and XDx32 players can maintain, along with the maximum sustained bitrate of those streams, is dependent on a number of factors.

4Kx42

Tests suggest that a 4Kx42 player can reliably maintain up to 50 simultaneous streams of a single 19Mb/s file (or 11 streams of different files with an average bitrate of approximately 16Mb/s). The network bandwidth will often be the limiting factor for streaming performance with a 4K player.

XDx32

An XDx32 player is usually limited by its 10/100 network interface. For example, the player can reliably stream four simultaneous streams of a single 19Mb/s file, while the fifth stream will begin to exhibit noticeable pauses.

Optimizing the Media Server

If during testing you are unable to replicate the level of performance outlined above, there are several steps you can take to ensure that there are no other bottlenecks in the streaming pipeline:

1. Ensure that the Media Server uses the `filesimple` source component and `udpsimple`/`rtpsimple` destination components whenever possible. The `file` and `udp`/`rtp` components consume more system resources than their simple counterparts.
2. Ensure the Media Server is not performing other tasks that are unrelated to streaming: downloading content, displaying content, etc.
3. Ensure the SD card or mSATA drive has a sufficient data read speed.

4. If you are attempting to stream a single large file, try using a file less than 200MB in size. This ensures that the file is read from the cache, rather than the SD card. Note that the filesize may need to be even smaller, depending on how memory is currently being consumed by other processes.