| | |
|---|---|
| MEDIASUBTYPE_Y41P | Y41P format data. A packed YUV format. A Y sample at every pixel, a U and V sample at every fourth pixel horizontally on each line; every vertical line sampled. Byte ordering (lowest first) is U0, Y0, V0, Y1, U4, Y2, V4, Y3, Y4, Y5, Y6, Y7, where the suffix 0 is the leftmost pixel and increasing numbers are pixels increasing left to right. Each 12-byte block is 8 image pixels. |
| MEDIASUBTYPE_YUY2 | YUY2 format data. Same as UYVY but with different pixel ordering. Byte ordering (lowest first) is Y0, U0, Y1, V0, Y2, U2, Y3, V2, Y4, U4, Y5, V4, where the suffix 0 is the leftmost pixel and increasing numbers are pixels increasing left to right. Each 4-byte block is 2 image pixels. |
| MEDIASUBTYPE_YVYU | YVYU format data. A packed YUV format. Same as UYVY but with different pixel ordering. Byte ordering (lowest first) is Y0, V0, Y1, U0, Y2, V2, Y3, U2, Y4, V4, Y5, U4, where the suffix 0 is the leftmost pixel and increasing numbers are pixels increasing left to right. Each 4-byte block is 2 image pixels. |
| MEDIASUBTYPE_UYVY | UYVY format data. A packed YUV format. A Y sample at every pixel, a U and V sample at every second pixel horizontally on each line; every vertical line sampled. Probably the most popular of the various YUV 4:2:2 formats. Byte ordering (lowest first) is U0, Y0, V0, Y1, U2, Y2, V2, Y3, U4, Y4, V4, Y5, where the suffix 0 is the leftmost pixel and increasing numbers are pixels increasing left to right. Each 4-byte block is 2 image pixels. |
| MEDIASUBTYPE_Y211 | YUV 211 format data. A packed YUV format. A Y sample at every second pixel, a U and V sample at every fourth pixel horizontally on each line; every vertical line sampled. Byte ordering (lowest first) is Y0, U0, Y2, V0, Y4, U4, Y6, V4, Y8, U8, Y10, V8, where the suffix 0 is the leftmost pixel and increasing numbers are pixels increasing left to right. Each 4-byte block is 4 image pixels. |
| MEDIASUBTYPE_CLJR | Cirrus Logic Jr YUV 411 format with less than 8 bits per Y, U, and V sample. Cinepak can produce it and Cirrus 5440 can produce an overlay with it. A Y sample at every pixel, a U and V sample at every fourth pixel horizontally on each line; every vertical line sampled. |
| MEDIASUBTYPE_IF09 | Indeo produced YVU9 format with additional information about differences from the last frame. 9.5 bits per pixel but reported as 9. |
| MEDIASUBTYPE_CPLA | Cinepak UYVY format. |
| MEDIASUBTYPE_MJPG | Motion JPEG (MJPG) compressed video. |
| MEDIASUBTYPE_TVMJ | TrueVision MJPG format. |
| MEDIASUBTYPE_WAKE | MJPG format produced by some cards. |
| MEDIASUBTYPE_CFCC | MJPG format produced by some cards. |
| MEDIASUBTYPE_IJPG | Intergraph JPEG format. |
| MEDIASUBTYPE_Plum | Plum MJPG format. |
| MEDIASUBTYPE_RGB1 | RGB 1 bit per pixel. Palettized. |
| MEDIASUBTYPE_RGB4 | RGB 4 bits per pixel. Palettized. |
| MEDIASUBTYPE_RGB8 | RGB 8 bits per pixel. Palettized. |

| | |
|---|---|
| MEDIASUBTYPE_RGB565 | 565 format of RGB, 16 bits per pixel. Uncompressed RGB samples. |
| MEDIASUBTYPE_RGB555 | 555 format of RGB, 16 bits per pixel. Uncompressed RGB samples. |
| MEDIASUBTYPE_RGB24 | RGB 24 bits per pixel. Uncompressed RGB samples. |
| MEDIASUBTYPE_RGB32 | RGB 32 bits per pixel. Uncompressed RGB samples. |
| MEDIASUBTYPE_Overlay | Video delivered using hardware overlay. |
| MEDIASUBTYPE_QTMovie | QT Specific compressions. |
| MEDIASUBTYPE_QTRpza | QT RPZA compressed data. |
| MEDIASUBTYPE_QTSmc | QT SMC compressed data. |
| MEDIASUBTYPE_QTRle | QT RLE compressed data. |
| MEDIASUBTYPE_QTJpeg | QT JPEG compressed data. |
| MEDIASUBTYPE_dvsd | Standard DV format. |
| MEDIASUBTYPE_dvhd | High Definition DV format. |
| MEDIASUBTYPE_dvsl | Long Play DV format. |
| MEDIASUBTYPE_MPEG1Packet | MPEG1 Video Packet. |
| MEDIASUBTYPE_MPEG1Payload | MPEG1 Video Payload. |
| MEDIASUBTYPE_VideoPort | Data is video port data, used with DVD. |

### Analog Video Media Types

The following analog video formats were introduced in ActiveMovie™ 1.0 but are currently not used. Instead, the **IAMAnalogVideoDecoder**, **IAMAnalogVideoEncoder** and IAMTVTuner interfaces use an enumeration called AnalogVideoStandard defined in Axextend.idl.

The following table describes the analog video media subtypes.

| **MEDIATYPE_AnalogVideo — Data is various formats of analog video, including standard NTSC, PAL, and SECAM formats.** |
|---|
| MEDIASUBTYPE_AnalogVideo_NTSC_M |
| MEDIASUBTYPE_AnalogVideo_PAL_B |
| MEDIASUBTYPE_AnalogVideo_PAL_D |
| MEDIASUBTYPE_AnalogVideo_PAL_G |
| MEDIASUBTYPE_AnalogVideo_PAL_H |
| MEDIASUBTYPE_AnalogVideo_PAL_I |
| MEDIASUBTYPE_AnalogVideo_PAL_M |
| MEDIASUBTYPE_AnalogVideo_PAL_N |
| MEDIASUBTYPE_AnalogVideo_SECAM_B |
| MEDIASUBTYPE_AnalogVideo_SECAM_D |
| MEDIASUBTYPE_AnalogVideo_SECAM_G |
| MEDIASUBTYPE_AnalogVideo_SECAM_H |
| MEDIASUBTYPE_AnalogVideo_SECAM_K |
| MEDIASUBTYPE_AnalogVideo_SECAM_K1 |
| MEDIASUBTYPE_AnalogVideo_SECAM_L |

# MPEG-1 Media Types

The following information summarizes the media types used by Microsoft® DirectShow™ for MPEG data.

MPEG-1 System Stream
Major type: MEDIATYPE_Stream

Minor type: MEDIASUBTYPE_MPEG1System

Format: None

Sample contents: BYTE stream; no alignment

MPEG-1 System Stream off Video CD
Major type: MEDIATYPE_Stream

Minor type: MEDIASUBTYPE_MPEG1VideoCD

Format: None

Sample contents: BYTE stream; no alignment

MPEG-1 Audio Packet
Major type: MEDIATYPE_Audio

Minor type: MEDIASUBTYPE_MPEG1Packet

Format: MPEG1WAVEFORMAT

Sample contents: Single MPEG-1 packet including packet header

MPEG-1 Audio payload
Major type: MEDIATYPE_Audio

Minor type: MEDIASUBTYPE_MPEG1Payload

Format: MPEG1WAVEFORMAT

Sample contents: Byte-aligned MPEG-1 audio data

MPEG-1 Video Packet
Major type: MEDIATYPE_Video

Minor type: MEDIASUBTYPE_MPEG1Packet

Format: VIDEOINFO + Video sequence header

Sample contents: Single MPEG-1 packet including packet header

MPEG-1 Video payload
Major type: MEDIATYPE_Video

Minor type: MEDIASUBTYPE_MPEG1Payload

Format: VIDEOINFO + Video sequence header

Sample contents: Byte-aligned MPEG-1 video data

MPEG-1 Native Video Stream
Major type: MEDIATYPE_Stream

Minor type: MEDIASUBTYPE_ MPEG1Video

Format: None

Sample contents: Array of video stream bytes (no system layer)

MPEG-1 Native Audio Stream
Major type: MEDIATYPE_Stream

Minor type: MEDIASUBTYPE_ MPEG1Audio

Format: None

Sample contents: Array of audio stream bytes (no system layer)

The various filters will support pins as follows:

| **Filter** | **Direction** | **Media type(s)** |
|---|---|---|
| System layer splitter | Input | MPEG-1 system stream |
| | | MPEG-1 system stream off Video CD |
| System layer splitter | Output | MPEG-1 Audio packet or MPEG-1 Audio data |
| System layer splitter | Output | MPEG-1 Video packet or MPEG-1 Video data |
| Software Audio CODEC | Input | MPEG-1 Audio data or MPEG-1 Audio packet |
| Software Video CODEC | Input | MPEG-1 Video data or MPEG-1 Video packet |
| Software Audio CODEC | Output | PCM audio mono or stereo, input sampling rate, input sampling rate divided by 2 or input sampling rate divided by 4 |
| Software Video CODEC | Output | Uncompressed video in Y41P, YUY2, UYVY, RGB24, RGB32, RGB565, RGB555 and RGB8 formats |

MPEG-1 Video packet and payload media types contain a complete sequence header so that data can be played from the middle of a file without needing a sequence header to initialize the video playback.

The video sequence header is appended to the video data type for MPEG video so that play can begin from the middle of a stream. The length of this field is up to 140 bytes (it includes the sequence header start code—0x000001B3—at the start and any quantization matrices found in the first sequence header encountered).

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

## Time Stamps

For more information about time stamps, see section 2.4.1 of ISO1-11172: "The packet header may contain decoding and/or presentation time stamps (DTS and PTS) that refer to the first access unit in the packet."

For MPEG_Stream major types, the start time is the byte position of the first byte, rated at 1 byte per second. The stop time is the byte position of the last byte. Thus, consecutive samples should have the stop time of the first packet equal to the start time of the next packet. For Video CD data, the origin of the medium must match the format of a video-CD file exposed by CDFS with the standard RIFF chunk at the start.

For MPEG video packet and payload types, the time stamp is the presentation time for the first video frame whose picture start code begins in the sample.

For MPEG audio packet and payload types, the time stamp is the presentation time for the first audio frame whose sync code starts in the sample.

It is assumed that packet and payload data without time stamps can be successfully prerolled by the handling filters.

Previous Home Topic Contents Index Next

## Sample Properties

MPEG samples have the following properties or notifications.

| | |
|---|---|
| Time stamp | Not all samples have start and stop times. The sample stop time for packet and payload data is not useful; it is usually set to the start time plus one. MPEG packet or payload data samples will have a start and stop time set if the system layer packet they are generated from had a valid PTS. |
| Discontinuity | If there is a break in the stream (for example, a gap in the real-time data, or an error in the data or after a seek), the Discontinuity property is set. This property is propagated from the MPEG-1 splitter to the stream handlers in the first sample sent after this property is set in a sample received. This also allows for a time-stamp discontinuity. |
| End Of Stream | This is not a sample property but a separate notification. When this is received, any buffered data must be forced through the decoder. Logically, any new data must then start with the Discontinuity property. |

Previous Home Topic Contents Index Next

# CLSIDs in DirectShow

Microsoft® DirectShow™ defines CLSIDs for many of its most-used components, such as filters and plug-in distributors. The CLSIDs are defined in Uuids.h. This article gives a brief description of the most common CLSIDs.

- Plug-in Distributor CLSIDs
- Cutlist CLSIDs
- Filter Category CLSIDs
- Filter and Filter Property Page CLSIDs

Previous Home Topic Contents Index Next

# Plug-in Distributor CLSIDs

This table lists the CLSIDs of the DirectShow plug-in distributors — COM objects that expose a control interface and implement it by calling the enumerator of the filter graph manager — finding which filters expose the control interface and communicating directly with those filters. The developer generally doesn't implement these interfaces.

| CLSID | Description |
|---|---|
| **CLSID_FilterGraph** | An object that builds filter graphs. This object implements the IFilterGraph interface. |
| **CLSID_CaptureGraphBuilder** | An object that builds capture graphs, preview graphs, and file compression graphs. This object implements the ICaptureGraphBuilder interface. |
| **CLSID_AMovie** | An object that performs as the filter graph manager. This object implements the IAMovie interface. |
| **CLSID_PersistMonikerPID** | An object that implements the **IPersistMoniker** interface, a standard COM interface that gives objects more control over the way they bind to their persistent data. |
| **CLSID_FilterMapper** | An object used by the filter graph manager to look up the properties of filters when they are loaded. This object implements the IFilterMapper interface. |
| **CLSID_SystemClock** | An object that implements the system reference clock in a filter graph. This object implements the IReferenceClock interface. |
| **CLSID_SeekingPassThru** | An object that implements the functionality of the CPosPassThru class. This object implements the IMediaSeeking and IMediaPosition interfaces. |

Previous Home Topic Contents Index Next

## Cutlist CLSIDs

This table lists the CLSIDs related to creating DirectShow cutlists, a collection of audio and video clips from different sources. Using the CutListGraphBuilderObject, the SimpleCutList object, and the VideoFileClip and AudioFileClip objects, an application can build a cutlist out of pieces of AVI and WAV files, and use the DirectShow Cutlist File Source filter to play it.

| CLSID | Description |
|---|---|
| **CLSID_AudioFileClip** | An object that represents an audio file clip in a cutlist. This object implements the IAMCutListElement interface. |
| **CLSID_VideoFileClip** | An object that represents a video file clip in a cutlist. This object implements IAMCutListElement. |
| **CLSID_SimpleCutList** | An object that represents a cutlist (a collection of cutlist elements, each with a relative time and duration). This object implements the IStandardCutList interface. |
| **CLSID_CutListGraphBuilder** | An object that represents a cutlist filter graph. This object implements the ICutListGraphBuilder interface. |
| **CLSID_MTXRiffs** | Cutlist File Source filter. |

Previous Home Topic Contents Index Next

## Filter Category CLSIDs

This table lists the CLSIDs for the DirectShow filter categories, seen in the Filter Graph Editor when you choose **Insert Filters** from the **Graph** menu. These categories can be used to enumerate the filters in a certain category.

| **CLSID** | **Description** |
|---|---|
| **CLSID_AudioInputDeviceCategory** or **CLSID_CWaveinClassManager** | Audio Capture Sources category |
| **CLSID_AudioCompressorCategory** or **CLSID_CAcmCoClassManager** | Audio Compressors category |
| **CLSID_AudioRendererCategory** or **CLSID_CWaveOutClassManager** | Audio Renderers category |
| **CLSID_LegacyAmFilterCategory** or **CLSID_CQzFilterClassManager** | DirectShow Filters category |
| **CLSID_MidiRendererCategory** or **CLSID_CMidiOutClassManager** | Midi Renderers category |
| **CLSID_VideoInputDeviceCategory** or **CLSID_CVidCapClassManager** | Video Capture Sources category |
| **CLSID_VideoCompressorCategory** or **CLSID_CIcmCoClassManager** | Video Compressors category |
| **CLSID_ActiveMovieCategories** | The seven categories of filters in DirectShow |

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

## Filter and Filter Property Page CLSIDs

This table lists the CLSIDs for the DirectShow filters, seen in the Filter Graph Editor when you choose **Insert Filters** from the **Graph** menu. It also contains the CLSIDs for the filter property pages.

| **CLSID** | **Description** |
|---|---|
| **CLSID_ACMWrapper** | ACM Audio Compressor filter |
| **CLSID_AudioRender** | Audio Renderer filter |
| **CLSID_AudioProperties** | Audio Renderer filter's property page |
| **CLSID_AVIDec** | AVI Decompressor filter |
| **CLSID_AviDest** | AVI MUX filter |
| **CLSID_AviMuxProptyPage** | AVI MUX filter's first property page |

| | |
|---|---|
| **CLSID_AviMUXProptyPage1** | AVI MUX filter's second property page |
| **CLSID_AviSplitter** | AVI Splitter filter |
| **CLSID_AviDoc** | AVI/WAV File Source filter |
| **CLSID_Colour** | Color Space Converter filter |
| **CLSID_MTXRiffs** | Cutlist File Source filter |
| **CLSID_DSoundRender** | DirectSound Audio Renderer filter, in the Audio Renderers category |
| **CLSID_DVMUX** | DV Muxer filter |
| **CLSID_DVMuxPropertyPage** | DV Muxer filter property page |
| **CLSID_DVSplitter** | DV Splitter filter |
| **CLSID_DVVideoCodec** | DV Video Decoder filter |
| **CLSID_DVDecPropertiesPage** | DV Video Decoder filter's property page |
| **CLSID_DVVideoEnc** | DV Video Encoder filter |
| **CLSID_DVEncPropertiesPage** | DV Video Encoder filter's property page |
| **CLSID_DVDNavigator** | DVD Navigator filter |
| **CLSID_AsyncReader** | File Source (Async) filter |
| **CLSID_URLReader** | File Source (URL) filter |
| **CLSID_FileWriter** | File Writer filter |
| **CLSID_ModexRenderer** | Full Screen Renderer filter |
| **CLSID_ModexProperties** | Full Screen Renderer filter's property page |
| **CLSID_InfTee** | Infinite Pin Tee filter |
| **CLSID_Line21Decoder** | Line21 Decoder filter |
| **CLSID_AVIMIDIRender** | MIDI Renderer filter |
| **CLSID_CMpegAudioCodec** | MPEG Audio Decoder filter |
| **CLSID_CMpegVideoCodec** | MPEG Video Decoder filter |
| **CLSID_MPEG1Splitter** | MPEG-1 Stream Splitter filter |
| **CLSID_OverlayMixer** | Overlay Mixer filter |
| **CLSID_TextRender** | Text Display filter |
| **CLSID_VfwCapture** | VFW Capture filter, in the Video Capture Sources category |
| **CLSID_CaptureProperties** | VFW Capture filter's property page |
| **CLSID_Dither** | VGA 16 Color Ditherer filter |
| **CLSID_VideoRenderer** | Video Renderer filter |
| **CLSID_AudioRecord** | WaveIn Audio Capture filter, in the Audio Capture Sources category |

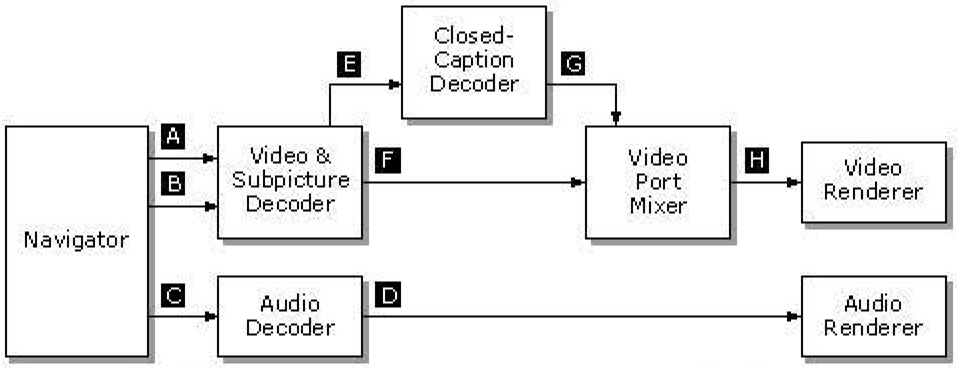
Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

# DirectShow DVD Support

This article outlines the DVD media types and formats. For the definitions of DirectShow-supported interlaced media types and picture aspect ratios, see VIDEOINFOHEADER2. MPEG2VIDEOINFO also contains relevant information.

The following diagram and tables specify the digital versatile disc (DVD) media types and formats supported by DirectShow.



| Connection | Major type | Subtype |
|---|---|---|
| A | MEDIATYPE_DVD_ENCRYPTED_PACK | MEDIASUBTYPE_MPEG2_VIDEO |
| B | MEDIATYPE_DVD_ENCRYPTED_PACK | MEDIASUBTYPE_DVD_SUBPICTURE |
| C | MEDIATYPE_DVD_ENCRYPTED_PACK | MEDIASUBTYPE_DOLBY_AC3 |
| C | MEDIATYPE_DVD_ENCRYPTED_PACK | MEDIASUBTYPE_DVD_LPCM_AUDIO |
| D | MEDIATYPE_AUDIO | NULL |
| E | MEDIATYPE_AUXLine21Data | MEDIASUBTYPE_Line21_GOPPacket |
| F | MEDIATYPE_VIDEO | MEDIASUBTYPE_VideoPort — see Note |
| G | MEDIATYPE_VIDEO | Standard video subtypes |
| H | MEDIATYPE_VIDEO | MEDIASUBTYPE_Overlay |

| Connection | Format block type | Format block structure |
|---|---|---|
| A | FORMAT_MPEG2Video | MPEG2VIDEOINFO |
| B | FORMAT_VideoInfo2 | MPEG2VIDEOINFO |
| C | FORMAT_WaveFormatEx | WAVEFORMATEX |
| C | FORMAT_WaveFormatEx | WAVEFORMATEX |
| D | FORMAT_WaveFormatEx | WAVEFORMATEX |
| E | FORMAT_VideoInfo2 | VIDEOINFOHEADER |
| F | FORMAT_VideoInfo2 | VIDEOINFOHEADER2 |
| G | FORMAT_VideoInfo2 | VIDEOINFOHEADER |
| H | FORMAT_VideoInfo | VIDEOINFOHEADER |

**Note:** DirectShow determines the appropriate video port pixel formats during transport phase negotiation with the IVPConfig interface.

Previous Home Topic Contents Index Next

# Country Codes and Channel to Frequency Mappings

The following information provides country codes, analog video standards, and channel to frequency mappings that are in use by most countries in the world. The IAMTVTuner interface uses this information to set and view analog broadcast or cable channels that will be viewed through a Microsoft® DirectShow™ TV Tuner filter.

**Contents of this article:**

**Country Codes**

The following table provides country code to integer mappings. These mappings are the same mappings used by the iCountry variable in Win.ini configuration file found in c:\Windows. The first column represents the actual country code. The second and third columns are cable and broadcast frequency lists, respectively, and the fourth column is the Analog Video Broadcast standard used in the country.

```
1,    F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,
                                          // United States of America
                                          // Anguilla
                                          // Antigua
                                          // Bahamas
                                          // Barbados
                                          // Bermuda
                                          // British Virgin Islands
                                          // Canada
                                          // Cayman Islands
                                          // Dominica
                                          // Dominican Republic
                                          // Grenada
                                          // Jamaica
                                          // Montserrat
                                          // Nevis
```

```
                                                // St. Kitts
                                                // St. Vincent and the Grenadines
                                                // Trinidad and Tobago
                                                // Turks and Caicos Islands
                                                // Barbuda
                                                // Puerto Rico
                                                // Saint Lucia
                                                // United States Virgin Islands

 2,   F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Canada (WIN.INI is bogus

 20,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_SECAM_B,     // Egypt
 212, F_FIX_CABLE, F_FIX_BROAD, AnalogVideo_SECAM_B,     // Morocco
 213, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Algeria
 216, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_SECAM_B,     // Tunisia
 218, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_SECAM_B,     // Libya
 220, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // Gambia
 221, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // Senegal Republic
 222, F_FIX_CABLE, F_FIX_BROAD, AnalogVideo_SECAM_B,     // Mauritania
 223, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_SECAM_K,     // Mali
 224, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_SECAM_K,     // Guinea
 225, F_FIX_CABLE, F_FIX_BROAD, AnalogVideo_SECAM_K,     // Ivory Coast
 226, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_SECAM_K,     // Burkina Faso
 227, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // Niger
 228, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // Togo
 229, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // Benin
 230, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_SECAM_B,     // Mauritius
 231, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Liberia
 232, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Sierra Leone
 233, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Ghana
 234, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Nigeria
 235, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Chad
 236, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Central African Republic
 237, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Cameroon
 238, F_FIX_CABLE, F_FIX_BROAD, AnalogVideo_FIX___,      // Cape Verde Islands
 239, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Sao Tome and Principe
 240, F_FIX_CABLE, F_FIX_BROAD, AnalogVideo_SECAM_B,     // Equatorial Guinea
 241, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // Gabon
 242, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_D,     // Congo
 243, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // Zaire
 244, F_FIX_CABLE, F_FIX_BROAD, AnalogVideo_PAL_I,       // Angola
 245, F_FIX_CABLE, F_FIX_BROAD, AnalogVideo_FIX___,      // Guinea-Bissau
 246, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Diego Garcia
 247, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Ascension Island
 248, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Seychelle Islands
 249, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Sudan
 250, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,,      // Rwanda
 251, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Ethiopia
 252, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Somalia
 253, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // Djibouti
 254, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Kenya
 255, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Tanzania
 256, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Uganda
 257, F_FIX_CABLE, F_FIX_BROAD, AnalogVideo_SECAM_K,     // Burundi
 258, F_FIX_CABLE, F_FIX_BROAD, AnalogVideo_PAL_B,       // Mozambique
 260, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Zambia
 261, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // Madagascar
 262, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // Reunion Island
 263, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Zimbabwe
 264, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_I,       // Namibia
 265, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Malawi
 266, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_I,       // Lesotho
 267, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_SECAM_K,     // Botswana
 268, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Swaziland
```

```
269, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_SECAM_K,     // Mayotte Island
269, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Comoros
27,  F_UK__CABLE, F_UK__BROAD, AnalogVideo_PAL_I,       // South Africa
290, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // St. Helena
291, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Eritrea
297, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Aruba
298, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Faroe Islands
299, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Greenland
30,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_SECAM_B,     // Greece
31,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Netherlands
32,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Belgium
33,  F_FRA_CABLE, F_FRA_BROAD, AnalogVideo_SECAM_L,     // France
34,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Spain
350, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Gibraltar
351, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Portugal
352, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Luxembourg
353, F_IRE_CABLE, F_IRE_BROAD, AnalogVideo_PAL_I,       // Ireland
354, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Iceland
355, F_ITA_CABLE, F_ITA_BROAD, AnalogVideo_PAL_B,       // Albania
356, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Malta
357, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Cyprus
358, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Finland
359, F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Bulgaria
36,  F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Hungary
370, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Lithuania
371, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_SECAM_D,     // Latvia
372, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Estonia
373, F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Moldova
374, F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Armenia
375, F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Belarus
376, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Andorra
377, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_SECAM_G,     // Monaco
378, F_ITA_CABLE, F_ITA_BROAD, AnalogVideo_PAL_B,       // San Marino
39,  F_ITA_CABLE, F_ITA_BROAD, AnalogVideo_PAL_B,       // Vatican City
380, F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Ukraine
381, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Yugoslavia
385, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Croatia
386, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Slovenia
387, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Bosnia and Herzegovina
389, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // F.Y.R.O.M. (Former Yugos
39,  F_ITA_CABLE, F_ITA_BROAD, AnalogVideo_PAL_B,       // Italy
40,  F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_PAL_D,       // Romania
41,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Switzerland
41,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Liechtenstein
42,  F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Czech Republic
42,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Slovak Republic
43,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Austria
44,  F_UK__CABLE, F_UK__BROAD, AnalogVideo_PAL_I,       // United Kingdom
45,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Denmark
46,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Sweden
47,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Norway
48,  F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Poland
49,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Germany
500, F_UK__CABLE, F_UK__BROAD, AnalogVideo_PAL_I,       // Falkland Islands
501, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Belize
502, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Guatemala
503, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // El Salvador
504, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Honduras
505, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Nicaragua
506, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Costa Rica
507, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Panama
508, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // St. Pierre and Miquelon
509, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Haiti
51,  F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Peru
```

```
52,  F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Mexico
53,  F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Cuba
53,  F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Guantanamo Bay
54,  F_USA_CABLE, F_USA_BROAD, AnalogVideo_PAL_N,       // Argentina
55,  F_USA_CABLE, F_USA_BROAD, AnalogVideo_PAL_M,       // Brazil
56,  F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Chile
57,  F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Colombia
58,  F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Venezuela
590, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // Guadeloupe
590, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // French Antilles
591, F_USA_CABLE, F_USA_BROAD, AnalogVideo_PAL_N,       // Bolivia
592, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // Guyana
593, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Ecuador
594, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // French Guiana
595, F_USA_CABLE, F_USA_BROAD, AnalogVideo_PAL_N,       // Paraguay
596, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // Martinique
597, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Suriname
598, F_USA_CABLE, F_USA_BROAD, AnalogVideo_PAL_N,       // Uruguay
599, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Netherlands Antilles
60,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Malaysia
61,  F_OZ__CABLE, F_OZ__BROAD, AnalogVideo_PAL_B,       // Australia
61,  F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Cocos-Keeling Islands
62,  F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Indonesia
63,  F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Philippines
64,  F_NZ__CABLE, F_NZ__BROAD, AnalogVideo_PAL_B,       // New Zealand
65,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Singapore
66,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Thailand
670, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Saipan Island
670, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Rota Island
670, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Tinian Island
671, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Guam
672, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Christmas Island
672, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Australian Antarctic Ter
672, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Norfolk Island
673, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Brunei
674, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Nauru
675, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Papua New Guinea
676, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Tonga
677, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Solomon Islands
678, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Vanuatu
679, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Fiji Islands
680, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Palau
681, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_SECAM_K,     // Wallis and Futuna Island
682, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Cook Islands
683, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Niue
684, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // American Samoa
685, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Western Samoa
686, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Kiribati Republic
687, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // New Caledonia
688, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Tuvalu
689, F_FOT_CABLE, F_FOT_BROAD, AnalogVideo_SECAM_K,     // French Polynesia
690, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Tokelau
691, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Micronesia
692, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Marshall Islands
7,   F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Russia
7,   F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Kazakhstan
7,   F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Kyrgyzstan
7,   F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Tajikistan
7,   F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Turkmenistan
7,   F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Uzbekistan

81,  F_JAP_CABLE, F_JAP_BROAD, AnalogVideo_NTSC_J,      // Japan

82,  F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Korea (South)
```

```
84,  F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Vietnam
850, F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Korea (North)
852, F_UK__CABLE, F_UK__BROAD, AnalogVideo_PAL_I,       // Hong Kong
853, F_UK__CABLE, F_UK__BROAD, AnalogVideo_PAL_I,       // Macau
855, F_USA_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Cambodia
856, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Laos
86,  F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_PAL_D,       // China
871, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // INMARSAT (Atlantic-East)
872, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // INMARSAT (Pacific)
873, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // INMARSAT (Indian)
874, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // INMARSAT (Atlantic-West)
880, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Bangladesh
886, F_USA_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Taiwan Region
90,  F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Turkey
91,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // India
92,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Pakistan
93,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Afghanistan
94,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Sri Lanka
95,  F_FIX_CABLE, F_USA_BROAD, AnalogVideo_NTSC_M,      // Myanmar
960, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Maldives
961, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_SECAM_B,     // Lebanon
962, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Jordan
963, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_SECAM_B,     // Syria
964, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_SECAM_B,     // Iraq
965, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Kuwait
966, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_SECAM_B,     // Saudi Arabia
967, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Yemen
968, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Oman
971, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // United Arab Emirates
972, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Israel
973, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Bahrain
974, F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_PAL_B,       // Qatar
975, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_FIX___,      // Bhutan
976, F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Mongolia
977, F_FIX_CABLE, F_USA_BROAD, AnalogVideo_PAL_B,       // Nepal
98,  F_WEU_CABLE, F_WEU_BROAD, AnalogVideo_SECAM_B,     // Iran
994, F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Azerbaijan
995, F_EEU_CABLE, F_EEU_BROAD, AnalogVideo_SECAM_D,     // Georgia
```

## Channel to Frequency Mappings for the U.S.

The following table provides the video carrier frequencies for the United States (U.S.) Broadcast is provided first, and then cable.

```
USA CABLE DATA

        1L,             // Lowest channel
        158L,           // Highest channel


        73250000L,      // 1    VHF-LO
        55250000L,      // 2    VHF-LO
        61250000L,      // 3
        67250000L,      // 4
        77250000L,      // 5
        83250000L,      // 6

       175250000L,      // 7    VHF-HI
       181250000L,      // 8
       187250000L,      // 9
       193250000L,      // 10
       199250000L,      // 11
       205250000L,      // 12
```

```
    211250000L,       // 13

    121250000L,       // 14   UHF
    127250000L,       // 15
    133250000L,       // 16
    139250000L,       // 17
    145250000L,       // 18
    151250000L,       // 19
    157250000L,       // 20
    163250000L,       // 21
    169250000L,       // 22
    217250000L,       // 23
    223250000L,       // 24
    229250000L,       // 25
    235250000L,       // 26
    241250000L,       // 27
    247250000L,       // 28
    253250000L,       // 29
    259250000L,       // 30
    265250000L,       // 31
    271250000L,       // 32
    277250000L,       // 33
    283250000L,       // 34
    289250000L,       // 35
    295250000L,       // 36
    301250000L,       // 37
    307250000L,       // 38
    313250000L,       // 39
    319250000L,       // 40
    325250000L,       // 41
    331250000L,       // 42
    337250000L,       // 43
    343250000L,       // 44
    349250000L,       // 45
    355250000L,       // 46
    361250000L,       // 47
    367250000L,       // 48
    373250000L,       // 49
    379250000L,       // 50
    385250000L,       // 51
    391250000L,       // 52
    397250000L,       // 53
    403250000L,       // 54
    409250000L,       // 55
    415250000L,       // 56
    421250000L,       // 57
    427250000L,       // 58
    433250000L,       // 59
    439250000L,       // 60
    445250000L,       // 61
    451250000L,       // 62
    457250000L,       // 63
    463250000L,       // 64
    469250000L,       // 65
    475250000L,       // 66
    481250000L,       // 67
    487250000L,       // 68
    493250000L,       // 69
    499250000L,       // 70
    505250000L,       // 71
    511250000L,       // 72
    517250000L,       // 73
    523250000L,       // 74
    529250000L,       // 75
```

```
535250000L,       // 76
541250000L,       // 77
547250000L,       // 78
553250000L,       // 79
559250000L,       // 80
565250000L,       // 81
571250000L,       // 82
577250000L,       // 83
583250000L,       // 84
589250000L,       // 85
595250000L,       // 86
601250000L,       // 87
607250000L,       // 88
613250000L,       // 89
619250000L,       // 90
625250000L,       // 91
631250000L,       // 92
637250000L,       // 93
643250000L,       // 94
 91250000L,       // 95    Discontinuity
 97250000L,       // 96
103250000L,       // 97
109250000L,       // 98
115250000L,       // 99
649250000L,       // 100   Discontinuity
655250000L,       // 101
661250000L,       // 102
667250000L,       // 103
673250000L,       // 104
679250000L,       // 105
685250000L,       // 106
691250000L,       // 107
697250000L,       // 108
703250000L,       // 109
709250000L,       // 110
715250000L,       // 111
721250000L,       // 112
727250000L,       // 113
733250000L,       // 114
739250000L,       // 115
745250000L,       // 116
751250000L,       // 117
757250000L,       // 118
763250000L,       // 119
769250000L,       // 120
775250000L,       // 121
781250000L,       // 122
787250000L,       // 123
793250000L,       // 124
799250000L,       // 125
805250000L,       // 126
811250000L,       // 127
817250000L,       // 128
823250000L,       // 129
829250000L,       // 130
835250000L,       // 131
841250000L,       // 132
847250000L,       // 133
853250000L,       // 134
859250000L,       // 135
865250000L,       // 136
871250000L,       // 137
877250000L,       // 138
883250000L,       // 139
```

```
        889250000L,      // 140
        895250000L,      // 141
        901250000L,      // 142
        907250000L,      // 143
        913250000L,      // 144
        919250000L,      // 145
        925250000L,      // 146
        931250000L,      // 147
        937250000L,      // 148
        943250000L,      // 149
        949250000L,      // 150
        955250000L,      // 151
        961250000L,      // 152
        967250000L,      // 153
        973250000L,      // 154
        979250000L,      // 155
        985250000L,      // 156
        991250000L,      // 157
        997250000L,      // 158

USA BROADCAST DATA

         2L,             // Lowest channel
         69L,            // Highest channel


         55250000L,      // 2    VHF-LO
         61250000L,      // 3
         67250000L,      // 4
         77250000L,      // 5
         83250000L,      // 6

        175250000L,      // 7    VHF-HI
        181250000L,      // 8
        187250000L,      // 9
        193250000L,      // 10
        199250000L,      // 11
        205250000L,      // 12
        211250000L,      // 13

        471250000L,      // 14   UHF
        477250000L,      // 15
        483250000L,      // 16
        489250000L,      // 17
        495250000L,      // 18
        501250000L,      // 19
        507250000L,      // 20
        513250000L,      // 21
        519250000L,      // 22
        525250000L,      // 23
        531250000L,      // 24
        537250000L,      // 25
        543250000L,      // 26
        549250000L,      // 27
        555250000L,      // 28
        561250000L,      // 29
        567250000L,      // 30
        573250000L,      // 31
        579250000L,      // 32
        585250000L,      // 33
        591250000L,      // 34
        597250000L,      // 35
        603250000L,      // 36
        609250000L,       // 37
        615250000L,       // 38
        621250000L,       // 39
        627250000L,       // 40
        633250000L,       // 41
        639250000L,       // 42
        645250000L,       // 43
        651250000L,       // 44
        657250000L,       // 45
        663250000L,       // 46
        669250000L,       // 47
        675250000L,       // 48
        681250000L,       // 49
        687250000L,       // 50
        693250000L,       // 51
        699250000L,       // 52
        705250000L,       // 53
        711250000L,       // 54
        717250000L,       // 55
        723250000L,       // 56
        729250000L,       // 57
        735250000L,       // 58
        741250000L,       // 59
        747250000L,       // 60
        753250000L,       // 61
        759250000L,       // 62
        765250000L,       // 63
        771250000L,       // 64
        777250000L,       // 65
        783250000L,       // 66
        789250000L,       // 67
        795250000L,       // 68
        801250000L,       // 69
```

**Channel to Frequency Mappings for Eastern Europe, China, and Russia**

The following table provides the video carrier frequencies for Eastern Europe, including China and Russia. Broadcast is provided first, and then cable.

```
EAST EUROPE/CHINA/RUSSIA CABLE DATA

         1L,              // Lowest channel
         57L,             // Highest channel

         49750000L,       // 1
         57250000L,       // 2
         65250000L,       // 3
         77250000L,       // 4
         85250000L,       // 5
        168250000L,       // 6
        176250000L,       // 7
        184250000L,       // 8
        192250000L,       // 9
        200250000L,       // 10
        208250000L,       // 11
        216250000L,       // 12
        471250000L,       // 13
        479250000L,       // 14
        487250000L,       // 15
        493250000L,       // 16
        503250000L,       // 17
        511250000L,       // 18
        519250000L,       // 19
        527250000L,       // 20
        535250000L,       // 21
        543250000L,       // 22
        551250000L,       // 23
        559250000L,       // 24
        607250000L,       // 25
        615250000L,       // 26
        623250000L,       // 27
        631250000L,       // 28
        639250000L,       // 29
        647250000L,       // 30
        655250000L,       // 31
        663250000L,       // 32
        671250000L,       // 33
        679250000L,       // 34
        687250000L,       // 35
        695250000L,       // 36
        703250000L,       // 37
        711250000L,       // 38
        719250000L,       // 39
        727250000L,       // 40
        735250000L,       // 41
        743250000L,       // 42
        751250000L,       // 43
        759250000L,       // 44
        767250000L,       // 45
        775250000L,       // 46
        783250000L,       // 47
        791250000L,       // 48
        799250000L,       // 49
        807250000L,       // 50
        815250000L,       // 51
        823250000L,       // 52
        831250000L,       // 53
        839250000L,       // 54
        847250000L,       // 55
        855250000L,       // 56
        863250000L,       // 57

 EAST EUROPE/CHINA/RUSSIA BROADCAST DATA

         1L,              // Lowest channel
         57L,             // Highest channel

         49750000L,       // 1
         57250000L,       // 2
         65250000L,       // 3
         77250000L,       // 4
         85250000L,       // 5
        168250000L,       // 6
        176250000L,       // 7
        184250000L,       // 8
        192250000L,       // 9
        200250000L,       // 10
        208250000L,       // 11
        216250000L,       // 12
        471250000L,       // 13
        479250000L,       // 14
        487250000L,       // 15
        493250000L,       // 16
        503250000L,       // 17
        511250000L,       // 18
        519250000L,       // 19
        527250000L,       // 20
        535250000L,       // 21
        543250000L,       // 22
        551250000L,       // 23
        559250000L,       // 24
        607250000L,       // 25
        615250000L,       // 26
        623250000L,       // 27
        631250000L,       // 28
        639250000L,       // 29
        647250000L,       // 30
        655250000L,       // 31
        663250000L,       // 32
        671250000L,       // 33
        679250000L,       // 34
        687250000L,       // 35
        695250000L,       // 36
        703250000L,       // 37
        711250000L,       // 38
        719250000L,       // 39
        727250000L,       // 40
        735250000L,       // 41
        743250000L,       // 42
        751250000L,       // 43
        759250000L,       // 44
        767250000L,       // 45
        775250000L,       // 46
        783250000L,       // 47
        791250000L,       // 48
        799250000L,       // 49
        807250000L,       // 50
        815250000L,       // 51
        823250000L,       // 52
        831250000L,       // 53
        839250000L,       // 54
        847250000L,       // 55
        855250000L,       // 56
        863250000L,       // 57
```

## Channel to Frequency Mappings for the French Overseas Territories

The following table provides the video carrier frequencies for the French Overseas Territories. Broadcast is provided first, and then cable.

```
FRENCH OVERSEAS TERRITORIES CABLE DATA
          1L,             // Lowest channel
          6L,             // Highest channel


        175250000L,       // 1
        183250000L,       // 2
        191250000L,       // 3
        199250000L,       // 4
        207250000L,       // 5
        215250000L,       // 6


FRENCH OVERSEAS TERRITORIES BROADCAST DATA
          1L,             // Lowest channel
          6L,             // Highest channel


        175250000L,       // 1
        183250000L,       // 2
        191250000L,       // 3
        199250000L,       // 4
        207250000L,       // 5
        215250000L,       // 6
```

## Channel to Frequency Mappings for France and CCIR L System Countries

The following table provides the video carrier frequencies for France and CCIR L System Countries. Broadcast is provided first, and then cable.

```
FRANCE CABLE DATA

         1L,              // Lowest channel
         69L,             // Highest channel

         47750000L,       // 1
         55750000L,       // 2
         60500000L,       // 3
         63750000L,       // 4
        176000000L,       // 5
        184000000L,       // 6
        192000000L,       // 7
        200000000L,       // 8
        208000000L,       // 9
        216000000L,       // 10
                0L,       // 11  Not used
                0L,       // 12  Not used
                0L,       // 13  Not used
                0L,       // 14  Not used
                0L,       // 15  Not used
                0L,       // 16  Not used
                0L,       // 17  Not used
                0L,       // 18  Not used
                0L,       // 19  Not used
                0L,       // 20  Not used
        471250000L,       // 21
        479250000L,       // 22
        487250000L,       // 23
        495250000L,       // 24
        503250000L,       // 25
        511250000L,       // 26
        519250000L,       // 27
        527250000L,       // 28
        535250000L,       // 29
        543250000L,       // 30
        551250000L,       // 31
        559250000L,       // 32
        567250000L,       // 33
        575250000L,       // 34
        583250000L,       // 35
        591250000L,       // 36
        599250000L,       // 37
        607250000L,       // 38
        615250000L,       // 39
        623250000L,       // 40
        631250000L,       // 41
        639250000L,       // 42
        647250000L,       // 43
        655250000L,       // 44
        663250000L,       // 45
        671250000L,       // 46
        679250000L,       // 47
        687250000L,       // 48
        695250000L,       // 49
        703250000L,       // 50
        711250000L,       // 51
        719250000L,       // 52
        727250000L,       // 53
        735250000L,       // 54
        743250000L,       // 55
        751250000L,       // 56
        759250000L,       // 57
        767250000L,       // 58
        775250000L,       // 59
        783250000L,       // 60
        791250000L,       // 61
        799250000L,       // 62
        807250000L,       // 63
        815250000L,       // 64
        823250000L,       // 65
        831250000L,       // 66
        839250000L,       // 67
        847250000L,       // 68
        855250000L,       // 69


 FRANCE BROADCAST DATA

         1L,              // Lowest channel
         69L,             // Highest channel


         47750000L,       // 1
         55750000L,       // 2
         60500000L,       // 3
         63750000L,       // 4
        176000000L,       // 5
        184000000L,       // 6
        192000000L,       // 7
        200000000L,       // 8
        208000000L,       // 9
        216000000L,       // 10
                0L,       // 11  Not used
                0L,       // 12  Not used
                0L,       // 13  Not used
                0L,       // 14  Not used
                0L,       // 15  Not used
                0L,       // 16  Not used
                0L,       // 17  Not used
                0L,       // 18  Not used
                0L,       // 19  Not used
                0L,       // 20  Not used
        471250000L,       // 21
        479250000L,       // 22
        487250000L,       // 23
        495250000L,       // 24
        503250000L,       // 25
        511250000L,       // 26
        519250000L,       // 27
        527250000L,       // 28
        535250000L,       // 29
        543250000L,       // 30
        551250000L,       // 31
        559250000L,       // 32
        567250000L,       // 33
        575250000L,       // 34
        583250000L,       // 35
        591250000L,      // 36
        599250000L,      // 37
        607250000L,      // 38
        615250000L,      // 39
        623250000L,      // 40
        631250000L,      // 41
        639250000L,      // 42
        647250000L,      // 43
        655250000L,      // 44
        663250000L,      // 45
        671250000L,      // 46
        679250000L,      // 47
        687250000L,      // 48
        695250000L,      // 49
        703250000L,      // 50
        711250000L,      // 51
        719250000L,      // 52
        727250000L,      // 53
        735250000L,      // 54
        743250000L,      // 55
        751250000L,      // 56
        759250000L,      // 57
        767250000L,      // 58
        775250000L,      // 59
        783250000L,      // 60
        791250000L,      // 61
        799250000L,      // 62
        807250000L,      // 63
        815250000L,      // 64
        823250000L,      // 65
        831250000L,      // 66
        839250000L,      // 67
        847250000L,      // 68
        855250000L,      // 69
```

**Channel to Frequency Mappings for Ireland**

The following table provides the video carrier frequencies for Ireland. Broadcast is provided first, and then cable.

```
IRELAND CABLE DATA

          1L,            // Lowest channel
          9L,            // Highest channel


         45750000L,      // 1
         53750000L,      // 2
         61750000L,      // 3
        175250000L,      // 4
        183250000L,      // 5
        191250000L,      // 6
        199250000L,      // 7
        207250000L,      // 8
        215250000L,      // 9


IRELAND BROADCAST DATA

          1L,            // Lowest channel
          9L,            // Highest channel
         45750000L,       // 1
         53750000L,       // 2
         61750000L,       // 3
        175250000L,       // 4
        183250000L,       // 5
        191250000L,       // 6
        199250000L,       // 7
        207250000L,       // 8
        215250000L,       // 9
```

## Channel to Frequency Mappings for Italy

The following table provides the video carrier frequencies for Italy. Broadcast is provided first, and then cable.

```
ITALY CABLE DATA

         2L,              // Lowest channel
         11L,             // Highest channel


         53750000L,       // 2    A
         62250000L,       // 3    B
         82250000L,       // 4    C
        175250000L,       // 5    D
        183750000L,       // 6    E
        192250000L,       // 7    F
        201250000L,       // 8    G
        210250000L,       // 9    H
        217250000L,       // 10   H1
        224250000L,       // 11   H2
```

## Channel to Frequency Mappings for Japan

The following table provides the video carrier frequencies for Japan. Broadcast is provided first, and then cable.

```
JAPAN CABLE DATA

         13L,             // Lowest channel
         63L,             // Highest channel


        109250000L,       // 13
        115250000L,       // 14
        121250000L,       // 15
        127250000L,       // 16
        133250000L,       // 17
        139250000L,       // 18
        145250000L,       // 19
        151250000L,       // 20
        157250000L,       // 21
        165250000L,       // 22
        223250000L,       // 23
        231250000L,       // 24
        237250000L,       // 25
        243250000L,       // 26
        249250000L,       // 27
        253250000L,       // 28
        259250000L,       // 29
        265250000L,       // 30
        271250000L,       // 31
        277250000L,       // 32
        283250000L,       // 33
        289250000L,       // 34
        295250000L,       // 35
        301250000L,       // 36
        307250000L,       // 37
        313250000L,       // 38
        319250000L,       // 39
        325250000L,       // 40
        331250000L,       // 41
        337250000L,       // 42
        343250000L,       // 43
        349250000L,       // 44
        355250000L,       // 45
        361250000L,       // 46
        367250000L,       // 47
        373250000L,       // 48
        379250000L,       // 49
        385250000L,       // 50
        391250000L,       // 51
        397250000L,       // 52
        403250000L,       // 53
        409250000L,       // 54
        415250000L,       // 55
        421250000L,       // 56
        427250000L,       // 57
        433250000L,       // 58
        439250000L,       // 59
        445250000L,       // 60
        451250000L,       // 61
        457250000L,       // 62
        463250000L,       // 63

JAPAN BROADCAST DATA

         1L,              // Lowest channel
         62L,             // Highest channel


         91250000L,       // 1
         97250000L,       // 2
        103250000L,       // 3
        171250000L,       // 4
        177250000L,       // 5
        183250000L,       // 6
        189250000L,       // 7
        193250000L,       // 8
        199250000L,       // 9
        205250000L,       // 10
        211250000L,       // 11
        217250000L,       // 12
        471250000L,       // 13
        477250000L,       // 14
        483250000L,       // 15
        489250000L,       // 16
        495250000L,       // 17
        501250000L,       // 18
        507250000L,       // 19
        513250000L,       // 20
        519250000L,       // 21
        525250000L,       // 22
        531250000L,       // 23
        537250000L,      // 24
        543250000L,      // 25
        549250000L,      // 26
        555250000L,      // 27
        561250000L,      // 28
        567250000L,      // 29
        573250000L,      // 30
        579250000L,      // 31
        585250000L,      // 32
        591250000L,      // 33
        597250000L,      // 34
        603250000L,      // 35
        609250000L,      // 36
        615250000L,      // 37
        621250000L,      // 38
        627250000L,      // 39
        633250000L,      // 40
        639250000L,      // 41
        645250000L,      // 42
        651250000L,      // 43
        657250000L,      // 44
        663250000L,      // 45
        669250000L,      // 46
        675250000L,      // 47
        681250000L,      // 48
        687250000L,      // 49
        693250000L,      // 50
        699250000L,      // 51
        705250000L,      // 52
        711250000L,      // 53
        717250000L,      // 54
        723250000L,      // 55
        729250000L,      // 56
        735250000L,      // 57
        741250000L,      // 58
        747250000L,      // 59
        753250000L,      // 60
        759250000L,      // 61
        765250000L,      // 62
```

## Channel to Frequency Mappings for New Zealand

The following table provides the video carrier frequencies for New Zealand. Broadcast is provided first, and then cable.

```
NEW ZEALAND CABLE DATA

          1L,            // Lowest channel
          9L,            // Highest channel


         45250000L,      // 1
         55250000L,      // 2
         62250000L,      // 3
        175250000L,      // 4
        182250000L,      // 5
        189250000L,      // 6
        196250000L,      // 7
        203250000L,      // 8
        210250000L,      // 9


NEW ZEALAND BROADCAST DATA
```

```
         1L,            // Lowest channel
         9L,            // Highest channel


        45250000L,      // 1
        55250000L,      // 2
        62250000L,      // 3
       175250000L,      // 4
       182250000L,      // 5
       189250000L,      // 6
       196250000L,      // 7
       203250000L,      // 8
       210250000L,      // 9
```

## Channel to Frequency Mappings for Australia

The following table provides the video carrier frequencies for Australia. Broadcast is provided first, and then cable.

```
AUSTRALIA CABLE DATA

         1L,            // Lowest channel
        14L,            // Highest channel


        46250000L,      // 0     (1)
        57250000L,      // 1     (2)
        64250000L,      // 2     (3)
        86250000L,      // 3     (4)
        95250000L,      // 4     (5)
       102250000L,      // 5     (6)
       138250000L,      // 5a    (7)
       175250000L,      // 6     (8)
       182250000L,      // 7     (9)
       189250000L,      // 8     (10)
       196250000L,      // 9     (11)
       209250000L,      // 10    (12)
       216250000L,      // 11    (13)
       223250000L,      // 12    (14)


AUSTRALIA BROADCAST DATA

         1L,            // Lowest channel
        14L,            // Highest channel


        46250000L,      // 0     (1)
        57250000L,      // 1     (2)
        64250000L,      // 2     (3)
        86250000L,      // 3     (4)
        95250000L,      // 4     (5)
       102250000L,      // 5     (6)
       138250000L,      // 5a    (7)
       175250000L,      // 6     (8)
       182250000L,      // 7     (9)
       189250000L,      // 8     (10)
       196250000L,      // 9     (11)
       209250000L,      // 10    (12)
       216250000L,      // 11    (13)
       223250000L,      // 12    (14)
```

## Channel to Frequency Mappings for the U.K., Ireland, S. Africa, and Hong Kong

The following table provides the video carrier frequencies for the United Kingdom (U.K.), Ireland, South Africa and Hong Kong. Broadcast is provided first, and then cable.

```
UK CABLE DATA

         2L,            // Lowest channel
        56L,            // Highest channel


        48250000L,      // 2
        55250000L,      // 3
        62250000L,      // 4
        69250000L,      // 5     S01
        76250000L,      // 6     S02
        83250000L,      // 7     S03
       105250000L,      // 8     S1
       112250000L,      // 9     S2
       119250000L,      // 10    S3
       126250000L,      // 11    S4
       133250000L,      // 12    S5
       140250000L,      // 13    S6
       147250000L,      // 14    S7
       154250000L,      // 15    S8
       161250000L,      // 16    S9
       168250000L,      // 17    S10
       175250000L,      // 18    E5
       182250000L,      // 19    E6
       189250000L,      // 20    E7
       196250000L,      // 21    E8
       203250000L,      // 22    E9
       210250000L,      // 23    E10
       217250000L,      // 24    E11
       224250000L,      // 25    E12
       231250000L,      // 26    S11
       238250000L,      // 27    S12
       245250000L,      // 28    S13
       252250000L,      // 29    S14
       259250000L,      // 30    S15
       266250000L,      // 31    S16
       273250000L,      // 32    S17
       280250000L,      // 33    S18
       287250000L,      // 34    S19
       294250000L,      // 35    S20
       303250000L,      // 36    S21
       311250000L,      // 37    S22
       319250000L,      // 38    S23
       327250000L,      // 39    S24
       335250000L,      // 40    S25
       343250000L,      // 41    S26
       351250000L,      // 42    S27
       359250000L,      // 43    S28
       367250000L,      // 44    S29
       375250000L,      // 45    S30
       383250000L,      // 46    S31
       391250000L,      // 47    S32
       399250000L,      // 48    S33
       407250000L,      // 49    S34
       415250000L,      // 50    S35
       423250000L,      // 51    S36
       431250000L,      // 52    S37
```

```
        439250000L,       // 53   S38
        447250000L,       // 54   S39
        455250000L,       // 55   S40
        463250000L,       // 56   S41


 UK BROADCAST DATA

         2L,              // Lowest channel
         69L,             // Highest channel


         45750000L,       // 2
         53750000L,       // 3
         61750000L,       // 4
        175250000L,       // 5
        183250000L,       // 6
        191250000L,       // 7
        199250000L,       // 8
        207250000L,       // 9
        215250000L,       // 10
                0L,       // 11  not used
                0L,       // 12  not used
                0L,       // 13  not used
                0L,       // 14  not used
                0L,       // 15  not used
                0L,       // 16  not used
                0L,       // 17  not used
                0L,       // 18  not used
                0L,       // 19  not used
                0L,       // 20  not used
        471250000L,       // 21
        479250000L,       // 22
        487250000L,       // 23
        495250000L,       // 24
        503250000L,       // 25
        511250000L,       // 26
        519250000L,       // 27
        527250000L,       // 28
        535250000L,       // 29
        543250000L,       // 30
        551250000L,       // 31
        559250000L,       // 32
        567250000L,       // 33
        575250000L,       // 34
        583250000L,       // 35
        591250000L,       // 36
        599250000L,       // 37
        607250000L,       // 38
        615250000L,       // 39
        623250000L,       // 40
        631250000L,       // 41
        639250000L,       // 42
        647250000L,       // 43
        655250000L,       // 44
        663250000L,       // 45
        671250000L,       // 46
        679250000L,       // 47
        687250000L,       // 48
        695250000L,       // 49
        703250000L,       // 50
        711250000L,       // 51
        719250000L,       // 52
        727250000L,       // 53
        735250000L,       // 54
        743250000L,       // 55
        751250000L,       // 56
        759250000L,       // 57
        767250000L,       // 58
        775250000L,       // 59
        783250000L,       // 60
        791250000L,       // 61
        799250000L,       // 62
        807250000L,       // 63
        815250000L,       // 64
        823250000L,       // 65
        831250000L,       // 66
        839250000L,       // 67
        847250000L,       // 68
        855250000L,       // 69
```

**Channel to Frequency Mappings for Western Europe**

The following table provides the video carrier frequencies for Western Europe. Broadcast is provided first, and then cable.

```
WESTERN EUROPE CABLE DATA

          2L,             // Lowest channel
         56L,             // Highest channel


         48250000L,       // 2
         55250000L,       // 3
         62250000L,       // 4
         69250000L,       // 5     S01
         76250000L,       // 6     S02
         83250000L,       // 7     S03
        105250000L,       // 8     S1
        112250000L,       // 9     S2
        119250000L,       // 10    S3
        126250000L,       // 11    S4
        133250000L,       // 12    S5
        140250000L,       // 13    S6
        147250000L,       // 14    S7
        154250000L,       // 15    S8
        161250000L,       // 16    S9
        168250000L,       // 17    S10
        175250000L,       // 18    E5
        182250000L,       // 19    E6
        189250000L,       // 20    E7
        196250000L,       // 21    E8
        203250000L,       // 22    E9
        210250000L,       // 23    E10
        217250000L,       // 24    E11
        224250000L,       // 25    E12
        231250000L,       // 26    S11
        238250000L,       // 27    S12
        245250000L,       // 28    S13
        252250000L,       // 29    S14
        259250000L,       // 30    S15
        266250000L,       // 31    S16
        273250000L,       // 32    S17
        280250000L,       // 33    S18
        287250000L,       // 34    S19
        294250000L,       // 35    S20
        303250000L,       // 36    S21
        311250000L,       // 37   S22
        319250000L,       // 38   S23
        327250000L,       // 39   S24
        335250000L,       // 40   S25
        343250000L,       // 41   S26
        351250000L,       // 42   S27
        359250000L,       // 43   S28
        367250000L,       // 44   S29
        375250000L,       // 45   S30
        383250000L,       // 46   S31
        391250000L,       // 47   S32
        399250000L,       // 48   S33
        407250000L,       // 49   S34
        415250000L,       // 50   S35
        423250000L,       // 51   S36
        431250000L,       // 52   S37
        439250000L,       // 53   S38
        447250000L,       // 54   S39
        455250000L,       // 55   S40
        463250000L,       // 56   S41

 WESTERN EUROPE BROADCAST DATA

         2L,              // Lowest channel
         69L,             // Highest channel


         48250000L,       // 2 Note: 2A is at 49.75
         55250000L,       // 3
         62250000L,       // 4
        175250000L,       // 5
        182250000L,       // 6
        189250000L,       // 7
        196250000L,       // 8
        203250000L,       // 9
        210250000L,       // 10
        217250000L,       // 11
        224250000L,       // 12

                0L,       // 13 Not used
                0L,       // 14 Not used
                0L,       // 15 Not used
                0L,       // 16 Not used
                0L,       // 17 Not used
                0L,       // 18 Not used
                0L,       // 19 Not used
                0L,       // 20 Not used

        471250000L,       // 21
        479250000L,       // 22
        487250000L,       // 23
        495250000L,       // 24
        503250000L,       // 25
        511250000L,       // 26
        519250000L,       // 27
        527250000L,       // 28
        535250000L,       // 29
        543250000L,       // 30
        551250000L,       // 31
        559250000L,       // 32
        567250000L,       // 33
        575250000L,       // 34
        583250000L,       // 35
        591250000L,       // 36
        599250000L,       // 37
        607250000L,       // 38
        615250000L,       // 39
        623250000L,       // 40
        631250000L,       // 41
        639250000L,       // 42
        647250000L,       // 43
        655250000L,       // 44
        663250000L,       // 45
        671250000L,       // 46
        679250000L,       // 47
        687250000L,       // 48
        695250000L,       // 49
        703250000L,       // 50
        711250000L,       // 51
        719250000L,       // 52
        727250000L,       // 53
        735250000L,       // 54
        743250000L,       // 55
        751250000L,       // 56
        759250000L,       // 57
        767250000L,       // 58
        775250000L,       // 59
        783250000L,       // 60
        791250000L,       // 61
        799250000L,       // 62
        807250000L,       // 63
        815250000L,       // 64
        823250000L,       // 65
        831250000L,       // 66
        839250000L,       // 67
        847250000L,       // 68
        855250000L,       // 69
```

## Reserved Identifiers

When compiling programs that include the Microsoft® DirectShow™ header files, it is important to be aware of the following identifiers and their meanings, if they are defined before the DirectShow headers are included.

These identifiers are, in effect, reserved except as described in the following list.

| | |
|---|---|
| **DEBUG** | Indicates a debug build. This enables many debug facilities, including message logging and assertion checking. |
| **PERF** | Indicates a performance (measurement) build. |
| **VFWROBUST** | Enables some pointer validation macros in a build of any kind. See ValidateReadPtr and related macros for more specifics. |

**Build Types**

A *retail build* is the smallest and fastest build, although perhaps not as robust as a debug build. A retail build is the default if neither a debug build nor a performance build is requested.

A *debug build* is generally the largest and slowest build. However, it provides better facilities for debugging new code and tracking problems. Defining the DEBUG identifier requests a debug build.

A *performance build* adds measurement capabilities to a retail build. Defining the PERF identifier requests a performance build. The performance build requires one additional file: Measure.dll.

**Caveat: Mixing Build Types**

If debug and retail binaries are mixed, the results are undefined. The results of mixing debug and retail object files in one binary are also undefined.

# Further Reading

**_CrtSetDbgFlag**
Retrieves and/or modifies the state of the _crtDbgFlag flag to control the allocation behavior of the debug heap manager (debug version only). See the Microsoft® Visual C++® documentation for more information.

**BitBlt**
The **BitBlt** function performs a bit-block transfer of the color data corresponding to a rectangle of pixels from the specified source device context into a destination device context. See the Microsoft Platform SDK documentation for more information.

**biHeight**
A data member of the BITMAPINFOHEADER structure that specifies the height of a bitmap, in pixels. If **biHeight** is positive, the bitmap is a bottom-up DIB (device-independent bitmap) and its origin is the lower left corner. If **biHeight** is negative, the bitmap is a top-down DIB and its origin is the upper left corner.

**biWidth**
A data member of the BITMAPINFOHEADER structure that specifies the width of a bitmap, in pixels.

**BSTR**
A 32-bit character pointer. See the Platform SDK documentation for more information.

**CAggDirectDraw::SetDisplayMode**
Sets the mode of the display-device hardware. See **IDirectDraw2::SetDisplayMode** in the Microsoft DirectX® SDK for more information.

**CAggDrawSurface::Blt**
Performs a bit block transfer. See **IDirectDrawSurface3::Blt** in the DirectX SDK for more information.

**CAggDrawSurface::GetDC**

Creates a GDI-compatible handle of a device context for the surface. See **IDirectDrawSurface3::GetDC** in the DirectX SDK for more information.

**CClassFactory**
A class that implements the **IClassFactory** interface. The **IClassFactory** interface contains two methods intended to deal with an entire class of objects, and so it is implemented on the class object for a specific class of objects (identified by a CLSID). The first method, **CreateInstance**, creates an uninitialized object of a specified CLSID, and the second, **LockServer**, locks the object's server in memory, enabling quicker creation of new objects. See the **IClassFactory** interface documentation in the Platform SDK.

**ClientToScreen**
Converts the client coordinates of a specified point to screen coordinates. See the Platform SDK documentation for more information.

**clipper**
See DirectDrawClipper.

**CoCreateInstance**
Creates a single uninitialized object of the class associated with a specified CLSID. See the Platform SDK documentation for more information.

**CoGetClassObject**
Provides a pointer to an interface on a class object associated with a specified CLSID. **CoGetClassObject** locates, and if necessary, dynamically loads the executable code required to do this. See the Platform SDK documentation for more information.

**CoInitialize**
Initializes the Component Object Model (COM) library. See the Platform SDK documentation for more information.

**COLORREF**
A 32-bit value used to specify an RGB color. See the Platform SDK documentation for more information.

**cooperative level**
Determines the top-level behavior of the application. See **IDirectDraw2::SetCooperativeLevel** in the DirectX SDK for more information.

**CoTaskMemAlloc**
Allocates a block of task memory in the same way that **IMalloc::Alloc** does. See the Platform SDK documentation for more information.

**CoTaskMemFree**
Frees a block of task memory previously allocated through a call to the **CoTaskMemAlloc** or **CoTaskMemRealloc** function. See the Platform SDK documentation for more information.

**cout**
A C++ object that controls insertions to the standard output as a byte stream. For more information, see the Run-Time Library Reference included in the Visual C++ Developer Studio documentation, or see other books that discuss the C and C++ programming languages.

**CreateDIBSection**
Creates a device-independent bitmap (DIB) that applications can write to directly. The function gives you a pointer to the location of the bitmap's bit values. You can supply a handle to a file mapping object that the function will use to create the bitmap, or you can let the operating system allocate the memory for the bitmap. See the Platform SDK documentation for more information.

**CreateEvent**
Creates a named or unnamed event object. See the Platform SDK documentation for more information.

**CreateFile**
Creates or opens various objects and returns a handle that can be used to access the object. See the Platform SDK documentation for more information.

**CreateWindow**
Creates an overlapped, pop-up, or child window. See the Platform SDK documentation for more information.
**CreateWindowEx**
Creates an overlapped, pop-up, or child window with an extended style; otherwise, this function is identical to the **CreateWindow** function. See the Platform SDK documentation for more information.
**CRITICAL_SECTION**
A critical section object, an object used to synchronize the threads of a single process. Only one thread at a time can own a critical-section object. See the Platform SDK documentation for more information.
**DDCAPS**
A structure that represents the capabilities of the hardware exposed through the Microsoft DirectDraw® object. See the DirectDraw documentation in the DirectX SDK for more information.
**DDCOLORCONTROL**
A structure that defines the color controls associated with a DirectDrawVideoPort object, an overlay surface, or a primary surface. See the DirectX SDK documentation for more information.
**DDEnumCallback**
An application-defined callback function for the **DirectDrawEnumerate** function. See the DirectDraw documentation in the DirectX SDK for more information.
**DDPIXELFORMAT**
A structure that describes the pixel format of a DirectDrawSurface object for the **IDirectDrawSurface3::GetPixelFormat** method. See the DirectDraw documentation in the DirectX SDK for more information.
**DDSCAPS**
A structure that defines the capabilities of a DirectDrawSurface object. This structure is part of the **DDCAPS** structure that is used to describe the capabilities of the DirectDraw object. See the DirectDraw documentation in the DirectX SDK for more information.
**DDSURFACEDESC**
A structure that contains a description of the surface to be created. This structure is passed to the **IDirectDraw2::CreateSurface** method. The relevant members differ for each potential type of surface. See the DirectDraw documentation in the DirectX SDK for more information.
**DDVIDEOPORTCONNECT**
A structure that describes a video port connection. See the DirectX SDK documentation for more information.
**DefWindowProc**
A member function that calls the default window procedure to provide default processing for any window messages that an application does not process. See the Platform SDK documentation for more information.
**DeleteCriticalSection**
A function that releases all resources used by an unowned critical section object. See the Platform SDK documentation for more information.
**DIBSECTION**
A structure that contains information about a device-independent bitmap created by calling the **CreateDIBSection** function. See the Platform SDK documentation for more information.
**DirectDraw**
DirectDraw® is a DirectX® SDK component that enables you to directly manipulate display memory, the hardware blitter, hardware overlay support, and flipping surface support. See the DirectX SDK for more information.
**DirectDrawClipper**
The object that DirectDraw uses to manage clip lists. A clip list is a series of rectangles

that describes the visible areas of the surface. A DirectDrawClipper object can be attached to any surface. A window handle can also be attached to a DirectDrawClipper object, and DirectDraw updates the DirectDrawClipper clip list with the clip list from the window as it changes. See the DirectX SDK for more information.

**DirectDrawCreate**
A function that creates an instance of a DirectDraw object. See the DirectX SDK for more information.

**DirectDrawEnumerate**
A function that enumerates the DirectDraw objects installed on the system. See the DirectX SDK for more information.

**DirectDrawSurface**
An object that represents an area in memory that holds data to be displayed on the monitor as images are moved to other surfaces. See "Surfaces" in the "DirectDraw Essentials" section of the DirectX SDK for more information.

**DispatchMessage**
A function that dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the GetMessage function. See the Platform SDK documentation for more information.

**DISPPARAMS**
A structure used by **IDispatch::Invoke** to contain the arguments passed to a method or property. See the Platform SDK documentation for more information.

**DllCanUnloadNow**
A function that determines whether the DLL that implements this function is in use. If not, the caller can safely unload the DLL from memory. See the Platform SDK documentation for more information.

**DllGetClassObject**
A function that is the entry point used by C++ file and stream handlers to create an instance of the handler. See the Platform SDK documentation for more information.

**DllRegisterServer**
A function that instructs an in-process server to create its registry entries for all classes supported in this server module. See the Platform SDK documentation for more information.

**DllUnregisterServer**
A function that instructs an in-process server to remove only those entries created through **DllRegisterServer**. See the Platform SDK documentation for more information.

**double**
The double keyword designates a 64-bit floating-point number. See the Platform SDK documentation for more information.

**DWORD**
A 32-bit unsigned integer or the address of a segment and its associated offset. See the Platform SDK documentation for more information.

**enum**
An enumerated type is a user-defined type consisting of a set of named constants called enumerators. See the Platform SDK documentation for more information.

**Err** object
A Visual Basic object that contains information about run-time errors. When a run-time error occurs, the Err object's properties are filled with information that identifies the error. To generate a run-time error in your Visual Basic code, use the **Raise** method. See Microsoft Visual Basic® documentation for more information.

**FAILED**
A function that provides a generic test for failure on any status value. Negative numbers indicate failure. See the Platform SDK for more information.

**FILETIME**
A structure that holds an unsigned 64-bit date and time value for a file. This value represents the number of 100-nanosecond units since the beginning of January 1, 1601.

See the Platform SDK documentation for more information.

**FOURCC**
A Four-Character Code used to identify Resource Interchange File Format (RIFF) chunks. A FOURCC is a 32-bit quantity represented as a sequence of one to four ASCII alphanumeric characters, padded on the right with blank characters. RIFF (Resource Interchange File Format) is a specification used to define standard formats for multimedia files and to prevent compatibility problems that often occur when file-format definitions change over time. Because each piece of data in the file is identified by a standard header, an application that does not recognize a given data element can skip over the unknown information. See the Platform SDK documentation for more information.

**GdiFlush**
A function that flushes the calling thread's current batch. Batching enhances drawing performance by minimizing the amount of time needed to call GDI drawing functions that return Boolean values.

**GetClassFile**
A function that supplies the CLSID associated with the given file name. See the Platform SDK documentation for more information.

**GetClientRect**
A function that retrieves the coordinates of a window's client area. See the Platform SDK documentation for more information.

**GetDDInterface**
An **IDirectDrawSurface3** method that retrieves an interface to the DirectDraw object that was used to create the surface. See the DirectX SDK for more information.

**GetLastError**
A function that returns the calling thread's last-error code value. See the Platform SDK documentation for more information.

**GetMessage**
The **GetMessage** function retrieves a message from the calling thread's message queue and places it in the specified structure.

**GetOpenFileName**
A function that creates an Open common dialog box that enables the user to specify the drive, directory, and the name of a file or set of files to open. See the Platform SDK documentation for more information.

**GetSystemPaletteEntries**
A function that retrieves a range of palette entries from the system palette that is associated with the specified device context. See the Platform SDK documentation for more information.

**GetWindowLong**
A function that retrieves information about the specified window. It also retrieves the 32-bit (long) value at the specified offset into a window's extra window memory. See the Platform SDK documentation for more information.

**HANDLE**
The handle of an object. See the Platform SDK documentation for more information.

**HBITMAP**
The handle of a bitmap. See the Platform SDK documentation for more information.

**Win32 HRESULT**
A value returned from a function call to an interface, consisting of a severity code, context information, a facility code, and a status code that describes the result. See the Platform SDK documentation for more information.

**IBindCtx**
An interface that provides access to a bind context, which is an object that stores information about a particular moniker binding operation. See the Platform SDK for more information.

**ICAbout**
A macro that notifies a video compression driver to display its About dialog box. See the Video

for Windows Development Kit version 1.1 for more information.

**ICConfigure**

A macro that notifies a video compression driver to display its configuration dialog box. See the Video for Windows Development Kit version 1.1 for more information.

**IClassFactory**

An interface that contains two methods intended to deal with an entire class of objects, and so is implemented on the class object for a specific class of objects (identified by a CLSID). The first method, **CreateInstance**, creates an uninitialized object of a specified CLSID, and the second, **LockServer**, locks the object's server in memory, allowing new objects to be created more quickly. See the Platform SDK documentation for more information.

**IClassFactory::CreateInstance**

A method that creates an uninitialized object. See the Platform SDK documentation for more information.

**ICSendMessage**

A function that sends a message to a compressor. See the Video for Windows Development Kit version 1.1 for more information.

**ICGetState**

A macro that queries a video compression driver to return its current configuration in a block of memory. You can use this macro or explicitly call the ICM_GETSTATE message. See the Platform SDK for more information.

**IDataObject**

An interface that specifies methods that enable data transfer and notification of changes in data. See the Platform SDK for more information.

**IDirectDraw**

Applications use the methods of this interface to create DirectDraw objects and work with system-level variables. See the DirectX SDK for more information.

**IDirectDraw::Compact**

A method that moves all of the pieces of surface memory on the display card to a contiguous block to make the largest single amount of free memory available. See **IDirectDraw2::Compact** in the DirectX SDK for more information.

**IDirectDraw::CreateClipper**

A method that creates a DirectDrawClipper object. See **IDirectDraw2::CreateClipper** the DirectX SDK for more information.

**IDirectDraw::CreatePalette**

A method that creates a DirectDrawPalette object for this DirectDraw object. See **IDirectDraw2::CreatePalette** in the DirectX SDK for more information.

**IDirectDraw::CreateSurface**

A method that creates a DirectDrawSurface object for the DirectDraw object. See **IDirectDraw2::CreateSurface** in the DirectX SDK for more information.

**IDirectDraw::DuplicateSurface**

A method that duplicates a DirectDrawSurface object. See **IDirectDraw2::DuplicateSurface** in the DirectX SDK for more information.

**IDirectDraw::EnumSurfaces**

A method that enumerates all of the existing or possible surfaces that meet the search criterion specified. See **IDirectDraw2::EnumSurfaces** in the DirectX SDK for more information.

**IDirectDraw::FlipToGDISurface**

A method that makes the surface that GDI writes to the primary surface. See **IDirectDraw2::FlipToGDISurface** in the DirectX SDK for more information.

**IDirectDraw::GetCaps**

A method that fills in the capabilities of the device driver for the hardware and the hardware-emulation layer (HEL). See **IDirectDraw2::GetCaps** in the DirectX SDK for more information.

**IDirectDraw::GetDisplayMode**

A method that retrieves the current display mode. See **IDirectDraw2::GetDisplayMode** in the DirectX SDK for more information.

**IDirectDraw::GetFourCCCodes**
A method that retrieves the FOURCC codes supported by the DirectDraw object. This method can also retrieve the number of codes supported. See **IDirectDraw2::GetFourCCCodes** in the DirectX SDK for more information.

**IDirectDraw::GetGDISurface**
A method that retrieves the DirectDrawSurface object that currently represents the surface memory that GDI is treating as the primary surface. See **IDirectDraw2::GetGDISurface** in the DirectX SDK for more information.

**IDirectDraw::GetMonitorFrequency**
A method that retrieves the frequency of the monitor being driven by the DirectDraw object. See **IDirectDraw2::GetMonitorFrequency** in the DirectX SDK for more information.

**IDirectDraw::EnumDisplayModes**
A method that enumerates all the display modes the hardware exposes through the DirectDraw object that are compatible with a provided surface description. See **IDirectDraw2::EnumDisplayModes** in the DirectX SDK for more information.

**IDirectDraw::GetScanLine**
A method that retrieves the scan line that is being drawn on the monitor. See **IDirectDraw2::GetScanLine** in the DirectX SDK for more information.

**IDirectDraw::GetVerticalBlankStatus**
A method that retrieves the status of the vertical blank. See **IDirectDraw2::GetVerticalBlankStatus** in the DirectX SDK for more information.

**IDirectDraw::Initialize**
A method that initializes the DirectDraw object that was created by using the COM **CoCreateInstance** function. See **IDirectDraw2::Initialize** in the DirectX SDK for more information.

**IDirectDraw::RestoreDisplayMode**
A method that resets the mode of the display device hardware for the primary surface to what it was before the **IDirectDraw2::SetDisplayMode** method was called. See **IDirectDraw2::RestoreDisplayMode** in the DirectX SDK for more information.

**IDirectDraw::SetCooperativeLevel**
A method that determines the application's top-level behavior. See **IDirectDraw2::SetCooperativeLevel** in the DirectX SDK for more information.

**IDirectDraw::SetDisplayMode**
A method that sets the mode of the display-device hardware. See **IDirectDraw2::SetDisplayMode** in the DirectX SDK for more information.

**IDirectDraw::WaitForVerticalBlank**
A method that helps the application synchronize itself with the vertical-blank interval. See **IDirectDraw2::WaitForVerticalBlank** in the DirectX SDK for more information.

**IDirectDraw2**
Applications use the methods of this interface to create DirectDraw objects and work with system-level variables. See the DirectX SDK for more information.

**IDirectDraw2::CreateSurface**
A method that creates a DirectDrawSurface object for the DirectDraw object. See the DirectX SDK for more information.

**IDirectDraw2::SetCooperativeLevel**
A method that determines the application's top-level behavior. See the DirectX SDK for more information.

**IDirectDrawClipper**
Applications use the methods of this interface to manage clip lists. See the DirectX SDK for more information.

**IDirectDrawClipper::SetHWnd**
A method that sets the window handle that will obtain the clipping information. See the

DirectX SDK for more information.

**IDirectDrawSurface**
An interface used to create DirectDrawSurface objects and work with system-level variables. See **IDirectDrawSurface3** in the DirectX SDK for more information.

**IDirectDrawSurface::AddAttachedSurface**
A method that attaches a surface to another surface. See **IDirectDrawSurface3::AddAttachedSurface** in the DirectX SDK for more information.

**IDirectDrawSurface::Blt**
A method that performs a bit block transfer. See **IDirectDrawSurface3::Blt** in the DirectX SDK for more information.

**IDirectDrawSurface::BltBatch**
A method that performs a sequence of **IDirectDrawSurface3::Blt** operations from several sources to a single destination. See **IDirectDrawSurface3::BltBatch** in the DirectX SDK for more information.

**IDirectDrawSurface::BltFast**
A method that performs a source copy blit or transparent blit by using a source color key or destination color key. See **IDirectDrawSurface3::BltFast** in the DirectX SDK for more information.

**IDirectDrawSurface::DeleteAttachedSurface**
A method that detaches two attached surfaces. The detached surface is not released. See **IDirectDrawSurface3::DeleteAttachedSurface** in the DirectX SDK for more information.

**IDirectDraw3::CreateClipper**
A method that creates a DirectDrawClipper object. See **IDirectDraw2::CreateClipper** in the DirectX SDK for more information.

**IDirectDrawSurface::EnumAttachedSurfaces**
A method that enumerates all the surfaces attached to a given surface. See **IDirectDrawSurface3::EnumAttachedSurfaces** in the DirectX SDK for more information.

**IDirectDrawSurface::EnumOverlayZOrders**
A method that enumerates the overlay surfaces on the specified destination. The overlays can be enumerated in front-to-back or back-to-front order. See **IDirectDrawSurface3::EnumOverlayZOrders** in the DirectX SDK for more information.

**IDirectDrawSurface::Flip**
A method that makes the surface memory associated with the DDSCAPS_BACKBUFFER surface become associated with the front-buffer surface. See **IDirectDrawSurface3::Flip** in the DirectX SDK for more information.

**IDirectDrawSurface::GetAttachedSurface**
A method that obtains the attached surface that has the specified capabilities. See **IDirectDrawSurface3::GetAttachedSurface** in the DirectX SDK for more information.

**IDirectDrawSurface::GetBltStatus**
A method that obtains the blitter status. See **IDirectDrawSurface3::GetBltStatus** in the DirectX SDK for more information.

**IDirectDrawSurface::GetCaps**
A method that retrieves the capabilities of the surface. These capabilities are not necessarily related to the capabilities of the display device. See **IDirectDrawSurface3::GetCaps** in the DirectX SDK for more information.

**IDirectDrawSurface::GetClipper**
A method that retrieves the DirectDrawClipper object associated with this surface. See **IDirectDrawSurface3::GetClipper** in the DirectX SDK for more information.

**IDirectDrawSurface::GetColorKey**
A method that retrieves the color key value for the DirectDrawSurface object. See **IDirectDrawSurface3::GetColorKey** in the DirectX SDK for more information.

**IDirectDrawSurface::GetDC**

A method that creates a GDI-compatible handle of a device context for the surface. See **IDirectDrawSurface3::GetDC** in the DirectX SDK for more information.

**IDirectDrawSurface::GetFlipStatus**
A method that indicates whether the surface has finished its flipping process. See **IDirectDrawSurface3::GetFlipStatus** in the DirectX SDK for more information.

**IDirectDrawSurface::GetOverlayPosition**
A method that retrieves the display coordinates of the surface. See **IDirectDrawSurface3::GetOverlayPosition** in the DirectX SDK for more information.

**IDirectDrawSurface::GetPalette**
A method that retrieves the DirectDrawPalette object associated with this surface and increments the reference count of the returned palette. See **IDirectDrawSurface3::GetPalette** in the DirectX SDK for more information.

**IDirectDrawSurface::GetPixelFormat**
A method that retrieves the color and pixel format of the surface. See **IDirectDrawSurface3::GetPixelFormat** in the DirectX SDK for more information.

**IDirectDrawSurface::GetSurfaceDesc**
A method that retrieves a **DDSURFACEDESC** structure that describes the surface in its current condition. See **IDirectDrawSurface3::GetSurfaceDesc** in the DirectX SDK for more information.

**IDirectDrawSurface::Initialize**
A method that initializes a DirectDrawSurface object. See **IDirectDrawSurface3::Initialize** in the DirectX SDK for more information.

**IDirectDrawSurface::IsLost**
A method that determines if the surface memory associated with a DirectDrawSurface object has been freed. See **IDirectDrawSurface3::IsLost** in the DirectX SDK for more information.

**IDirectDrawSurface::Lock**
A method that obtains a pointer to the surface memory. See **IDirectDrawSurface3::Lock** in the DirectX SDK for more information.

**IDirectDrawSurface::ReleaseDC**
A method that releases the handle of a device context previously obtained by using the **IDirectDrawSurface3::GetDC** method. See **IDirectDrawSurface3::ReleaseDC** in the DirectX SDK for more information.

**IDirectDrawSurface::Restore**
A method that restores a surface that has been lost. This occurs when the surface memory associated with the DirectDrawSurface object has been freed. See **IDirectDrawSurface3::Restore** in the DirectX SDK for more information.

**IDirectDrawSurface::SetClipper**
A method that attaches a DirectDrawClipper object to a DirectDrawSurface object. See **IDirectDrawSurface3::SetClipper** in the DirectX SDK for more information.

**IDirectDrawSurface::SetColorKey**
A method that sets the color key value for the DirectDrawSurface object if the hardware supports color keys on a per surface basis. See **IDirectDrawSurface3::SetColorKey** in the DirectX SDK for more information.

**IDirectDrawSurface::SetOverlayPosition**
A method that changes the display coordinates of an overlay surface. See **IDirectDrawSurface3::SetOverlayPosition** in the DirectX SDK for more information.

**IDirectDrawSurface::SetPalette**
A method that attaches the specified DirectDrawPalette object to a surface. The surface uses this palette for all subsequent operations. The palette change happens immediately, without regard to refresh timing. See **IDirectDrawSurface3::SetPalette** in the DirectX SDK for more information.

**IDirectDrawSurface::Unlock**
A method that notifies DirectDraw that the direct surface manipulations are complete. See **IDirectDrawSurface3::Unlock** in the DirectX SDK for more information.

**IDirectDrawSurface::UpdateOverlay**
A method that repositions or modifies the visual attributes of an overlay surface. These surfaces must have the DDSCAPS_OVERLAY value set. See **IDirectDrawSurface3::UpdateOverlay** in the DirectX SDK for more information.

**IDirectDrawSurface::UpdateOverlayDisplay**
A method that repaints the rectangles in the dirty rectangle list of all active overlays. See **IDirectDrawSurface3::UpdateOverlayDisplay** in the DirectX SDK for more information.

**IDirectDrawSurface::UpdateOverlayZOrder**
A method that sets the z-order of an overlay. See **IDirectDrawSurface3::UpdateOverlayZOrder** in the DirectX SDK for more information.

**IDirectDrawSurface3::Blt**
A method that performs a bit block transfer. See the DirectX SDK for more information.

**IDirectSound**
Applications use the methods of this interface to create DirectSound objects and set up the environment. See the DirectX SDK for more information.

**IDirectSoundBuffer**
Applications use the methods of this interface to create DirectSoundBuffer objects and set up the environment. See the DirectX SDK for more information.

**IDispatch**
An interface that exposes objects, methods, and properties to Automation programming tools and other applications. A dual interface derives from **IDispatch** and uses only Automation-compatible types. Like the **IDispatch** interface, a dual interface supports early and late binding. However, a dual interface differs in that it also supports vtable binding. See the Platform SDK documentation for more information.

**IDispatch::GetIDsOfNames**
A method that maps a single member and an optional set of argument names to a corresponding set of integer DISPIDs (dispatch identifiers), which can be used on subsequent calls to **IDispatch::Invoke**. See the Platform SDK documentation for more information.

**IDispatch::GetTypeInfo**
A method that retrieves the type information for an object, which can then be used to get the type information for an interface. See the Platform SDK documentation for more information.

**IDispatch::GetTypeInfoCount**
A method that retrieves the number of type information interfaces that an object provides (either 0 or 1). See the Platform SDK documentation for more information.

**IDispatch::Invoke**
A method that provides access to properties and methods exposed by an object. See the Platform SDK documentation for more information.

**IEnumMoniker**
An interface used to enumerate the components of a moniker or to enumerate the monikers in a table of monikers. See the Platform SDK documentation for more information.

**IEnumVARIANT**
A dispatch interface that provides a way to iterate over collection objects. See the Platform SDK documentation for more information.

**IEnumXXXX**
A set of enumeration interfaces that enable you to enumerate the number of items of a given type that an object maintains. There is one interface for each type of item. To use these interfaces, the client asks an object that maintains a collection of items to create an enumerator object. The interface on the enumeration object is one of the enumeration interfaces, all of which have a name of the form IEnumItem_name. The only difference among the enumeration interfaces is what they enumerate. There must be a separate

enumeration interface for each type of item enumerated. All have the same set of methods, and are used in the same way. See the Platform SDK documentation for more information.

**IMoniker**
An interface containing methods that enable you to use a moniker object, which contains information that uniquely identifies a COM object. An object that has a pointer to the moniker object's **IMoniker** interface can locate, activate, and get access to the identified object without having any other specific information on where the object is actually located in a distributed system. See the COM documentation in the Platform SDK for more information.

**IMoniker::BindToStorage**
A method that retrieves an interface pointer to the storage that contains the object identified by the moniker. Unlike **IMoniker::BindToObject**, this method does not activate the object identified by the moniker. See the COM documentation in the Platform SDK for more information.

**IMoniker::BindToObject**
A method that uses the moniker to bind to the object it identifies. The binding process involves finding the object, putting it into the running state if necessary, and supplying the caller with a pointer to a specified interface on the identified object. See the COM documentation in the Platform SDK for more information.

**InitializeCriticalSection**
A function that initializes a critical section object, which is an object used to synchronize the threads of a single process. Only one thread at a time can own a critical-section object. See the Platform SDK documentation for more information.

**IPersist**
An interface with one method, **GetClassID**, which is designed to supply the CLSID of an object that can be stored persistently in the system. You must implement the single method of **IPersist** in implementing any one of the other persistence interfaces: **IPersistStorage**, **IPersistStream**, or **IPersistFile**. You can use **IPersist** when all that is required is to obtain the CLSID of a persistent object, as it is used in marshaling. See the Platform SDK documentation for more information.

**IPersistFile**
An interface that provides methods that permit an object to be loaded from or saved to a disk file, rather than a storage object or stream. Typically, for example, you would implement **IPersistFile** on a linked object. See the Platform SDK documentation for more information.

**IPersistMediaPropertyBag**
Documentation for this interface is identical to documentation for **IPersistPropertyBag** found in the COM documentation in the Platform SDK except for the following additions:

1.) The **Load** method can return STG_E_ACCESSDENIED to indicate that the object is read-only (the AVI parser, for example does this).

2.) The **Save** method can return E_NOTIMPL. **IPersistPropetyBag::Save** does not permit this.

**IPersistPropertyBag**
An interface that works in conjunction with **IPropertyBag** and **IErrorLog** to define an individual property-based persistence mechanism. See the COM documentation in the Platform SDK for more information.

**IPersistPropertyBag::Load**
A method called by the container to load the control's properties. See the COM documentation in the Platform SDK for more information.

**IPersistStream**
An interface that provides methods for saving and loading objects that use a simple

serial stream for their storage needs. See the Platform SDK documentation for more information.

**IPersistStream::GetSizeMax**
A method that returns the size, in bytes, of the stream needed to save the object. See the Platform SDK documentation for more information.

**IPersistStream::IsDirty**
A method that checks the object for changes since it was last saved. See the Platform SDK documentation for more information.

**IPersistStream::Load**
A method that initializes an object from the stream where it was previously saved. See the Platform SDK documentation for more information.

**IPersistStream::Save**
A method that saves an object into the specified stream and indicates whether the object should reset its dirty flag. See the Platform SDK documentation for more information.

**IPropertyBag**
An interface that provides an object with a property bag in which the object can persistently save its properties. See the Platform SDK documentation for more information.

**IPropertyPage**
An interface that provides the main features of a property page object that manages a particular page within a property sheet. See the Platform SDK documentation for more information.

**IPropertyPage::Active**
A method that creates the dialog box window for the property page. See the Platform SDK documentation for more information.

**IPropertyPage::Apply**
A method that applies current property page values to underlying objects specified through the **SetObjects** method. See the Platform SDK documentation for more information.

**IPropertyPage::Deactivate**
A method that destroys the window created with the **Activate** method. See the Platform SDK documentation for more information.

**IPropertyPage::GetPageInfo**
A method that retrieves information about the property page. See the Platform SDK documentation for more information.

**IPropertyPage::Help**
A method that invokes Help in response to end-user request. See the Platform SDK documentation for more information.

**IPropertyPage::IsPageDirty**
A method that indicates whether the property page has changed since activated or since the most recent call to the **Apply** method. See the Platform SDK documentation for more information.

**IPropertyPage::Move**
A method that positions and resizes the property page dialog box within the frame. See the Platform SDK documentation for more information.

**IPropertyPage::SetObjects**
A method that provides the property page with an array of **IUnknown** pointers for objects associated with this property page. See the Platform SDK documentation for more information.

**IPropertyPage::SetPageSite**
A method that initializes a property page and provides the page with a pointer to the IPropertyPageSite interface, through which the property page communicates with the property frame. See the Platform SDK documentation for more information.

**IPropertyPage::Show**
A method that makes the property page dialog box visible or invisible. See the Platform

SDK documentation for more information.

**IPropertyPage::TranslateAccelerator**
A method that provides a pointer to a **MSG** structure that specifies a keystroke to process. See the Platform SDK documentation for more information.

**IPropertyPageSite**
An interface that provides the main features for a property page site object. See the Platform SDK documentation for more information.

**IsBadReadPtr**
A Win32 function that verifies that the calling process has read access to the specified range of memory. See the Platform SDK documentation for more information.

**ISpecifyPropertyPages**
An interface that indicates that an object supports property pages. See the Platform SDK documentation for more information.

**ISpecifyPropertyPages::GetPages**
A method that fills an array of CLSIDs for each property page that can be displayed in this object's property sheet. See the Platform SDK COM documentation for more information.

**IStorage::OpenStream**
A method that opens an existing stream object within this storage object using the specified access permissions in the *grfMode* parameter. See the Platform SDK documentation for more information.

**IStream**
An interface that supports reading and writing data to stream objects. See the Platform SDK documentation for more information.

**ITypeInfo**
An interface typically used for reading information about objects. For example, an object browser tool can use **ITypeInfo** to extract information about the characteristics and capabilities of objects from type libraries. See the Platform SDK documentation for more information.

**LoadLibrary**
A function that maps the specified executable module into the address space of the calling process. See the Platform SDK documentation for more information.

**long**
A keyword that designates a 32-bit integer. See the Platform SDK documentation for more information.

**Long**
The Visual Basic 32-bit integer. See Visual Basic documentation for more information.

**LONG**
A 32-bit signed integer. See the Platform SDK documentation for more information.

**LONGLONG**
A 64-bit signed integer. See the Platform SDK documentation for more information.

**LPCTSTR**
A pointer to a constant null-terminated Unicode or Windows character string. See the Platform SDK documentation for more information.

**LPDDSURFACEDESC**
A LONG pointer to a **DDSURFACEDESC** structure that contains a description of the surface to be created. See the DirectDraw documentation in the DirectX SDK for more information.

**LPSTR**
A pointer to a null-terminated Windows character string. See the Platform SDK documentation for more information.

**LPWSTR**
A pointer to a null-terminated Unicode character string. See the Platform SDK documentation for more information.

**LRESULT**

A 32-bit value returned from a window procedure or callback function. See the Platform SDK documentation for more information.

**memcmp**

A C function that compares characters in two buffers. For more information, see the Run-Time Library Reference included in the Visual C++ Developer Studio documentation, or see other books that discuss the C and C++ programming languages.

**MainAVIHeader**

A structure that contains global information for the entire AVI file. See the Platform SDK documentation for more information.

**MoveWindow**

A function that changes the position and dimensions of the specified window. See the Platform SDK documentation for more information.

**MSG**

A structure that contains message information from a thread's message queue. See the Platform SDK documentation for more information.

**MsgWaitForMultipleObjects**

A function that determines whether the wait criteria have been met. See the Platform SDK documentation for more information.

**MultiByteToWideChar**

A function that maps a character string to a wide-character (Unicode) string. See the Platform SDK documentation for more information.

**Number** property

A property used to determine the nature of an error that occurred on a remote server or in the ODBC (Open Database Connectivity) interface. See Visual Basic documentation for more information.

**OleCreatePropertyFrame**

A function that invokes a new property frame; that is, a property sheet dialog box, whose parent is *hwndOwner*, where the dialog is positioned at the point (x,y) in the parent window and has the caption *lpszCaption*. See the Platform SDK documentation for more information.

**OPENFILENAME**

A structure that contains information that the **GetOpenFileName** and **GetSaveFileName** functions use to initialize an Open or Save As common dialog box. See the Platform SDK documentation for more information.

**OutputDebugString**

A function that sends a string to the debugger for the current application. See the Platform SDK documentation for more information.

**PALETTEENTRY**

A structure that specifies the color and usage of an entry in a logical color palette. A logical palette is defined by a **LOGPALETTE** structure. See the Platform SDK documentation for more information.

**PCMWAVEFORMAT**

A structure that describes the data format for PCM waveform-audio data. See the Platform SDK documentation for more information.

**PeekMessage**

A function that checks a thread message queue for a message and places the message (if any) in the specified structure. See the Platform SDK documentation for more information.

**PostMessage**

A function that places (posts) a message in the message queue associated with the thread that created the specified window, and then returns without waiting for the thread to process the message. See the Platform SDK documentation for more information.

**printf**

A C function that prints formatted output to the standard output stream. For more information, see the Run-Time Library Reference included in the Visual C++ Developer

Studio documentation, or see other books that discuss the C and C++ programming languages.

**PROPPAGEINFO**
A structure that contains parameters used to describe a property page to a property frame. See the Platform SDK documentation for more information.

**RECT**
A structure that defines the coordinates of the upper-left and lower-right corners of a rectangle. See the Platform SDK documentation for more information.

**ReleaseSemaphore**
A function that increases the count of the specified semaphore object by a specified amount. See the Platform SDK documentation for more information.

**RGBQUAD**
A structure that describes a color consisting of relative intensities of red, green, and blue. See the Platform SDK documentation for more information.

**ScaleHeight**
A property that returns or sets the number of units for the vertical measurement of the interior of an object when using graphics methods or when positioning controls. See Visual Basic documentation for more information.

**ScaleWidth**
A property that retrieves or sets the number of units for the horizontal measurement of an object's interior when using graphics methods or when positioning controls. See Visual Basic documentation for more information.

**SendMessage**
A function that sends the specified message to a window or windows. See the Platform SDK documentation for more information.

**SetDIBColorTable**
A function that sets RGB (red, green, blue) color values in a range of entries in the color table of the device-independent bitmap (DIB) that is selected into a specified device context. See the Platform SDK documentation for more information.

**SetDIBitsToDevice**
A function that sets the pixels in the specified rectangle on the device that is associated with the destination device context using color data from a device-independent bitmap (DIB). See the Platform SDK documentation for more information.

**SetDlgItemText**
A function that sets the title or text of a control in a dialog box. See the Platform SDK documentation for more information.

**SetParent**
A function that changes the parent window of the specified child window. See the Platform SDK documentation for more information.

**SetWindowLong**
A function that changes an attribute of the specified window. The function also sets a 32-bit (long) value at the specified offset into the extra window memory of a window. See the Platform SDK documentation for more information.

**ShowWindow**
A function that sets the specified window's show state. See the Platform SDK documentation for more information.

**SIZE**
A structure that specifies the width and height of a rectangle. See the Platform SDK documentation for more information.

**sscanf**
A C function that reads formatted data from a string. For more information, see the Run-Time Library Reference included in the Visual C++ Developer Studio documentation, or other books that discuss the C and C++ programming languages.

**StgOpenStorage**
A function that opens an existing root storage object in the file system. You can use this

function to open compound files, but you can't use it to open directories, files, or summary catalogs. See the Platform SDK documentation for more information.

**StretchBlt**
A function that copies a bitmap from a source rectangle into a destination rectangle, stretching or compressing the bitmap to fit the dimensions of the destination rectangle, if necessary. Windows stretches or compresses the bitmap according to the stretching mode currently set in the destination device context. See the Platform SDK documentation for more information.

**StretchDIBits**
A function that copies the color data for a rectangle of pixels in a device-independent bitmap (DIB) to the specified destination rectangle. If the destination rectangle is larger than the source rectangle, this function stretches the rows and columns of color data to fit the destination rectangle. If the destination rectangle is smaller than the source rectangle, this function compresses the rows and columns by using the specified raster operation. See the Platform SDK documentation for more information.

**SUCCEEDED**
A function that provides a generic test for success on any status value. Non-negative numbers indicate success. See the Platform SDK documentation for more information.

**SysAllocString**
A function that allocates a new string and copies the passed string into it. See the Platform SDK documentation for more information.

**SysFreeString**
A function that frees a previously allocated string. See the Platform SDK documentation for more information.

**timeBeginPeriod**
A function that sets the minimum timer resolution for an application or device driver. See the Platform SDK documentation for more information.

**timeGetTime**
A function that retrieves the system time, in milliseconds. The system time is the time elapsed since Windows was started. See the Platform SDK documentation for more information.

**timeSetEvent**
A function that starts a specified timer event. The multimedia timer runs in its own thread. After the event is activated, it calls the specified callback function. See the Platform SDK documentation for more information.

**TranslateMessage**
A function that translates virtual-key messages into character messages. The character messages are posted to the calling thread's message queue, to be read the next time the thread calls the GetMessage or PeekMessage function. See the Platform SDK documentation for more information.

**TXTDT_MG**
A structure that can contain text descriptions of the video. See Section 4.1.6 and Annex A of the DVD-Video specification for more information. To obtain a copy of the specification, contact Toshiba Corporation at 1-1, Shibaura 1-Chrome, Minato-ku, Tokyo 105-01, Japan, Tel. +81-3-5444-9580, Fax. +81-3-5444-9430.

**videoDialog**
A function that displays a dialog box used to set configuration parameters for a video capture device driver. See the Video for Windows Development Kit version 1.1 for more information.

**videoMessage**
A function that sends messages to a video capture device driver. See the Video for Windows Development Kit version 1.1 for more information.

**WaitForMultipleObjects**
A Win32 function that determines whether wait criteria have been met. If the criteria have not been met, the calling thread enters a wait state. The function returns when any

one or all of the specified objects are in the signaled state, or when the time-out interval elapses. See the Platform SDK documentation for more information.

**WaitForSingleObject**
: A Win32 function that checks the current state of the specified object. If the object's state is nonsignaled, the calling thread enters a wait state. The function returns when returns when the specified object is in the signaled state, or when the time-out interval elapses. See the Platform SDK documentation for more information.

**WAVEFORMAT**
: A structure that describes the format of waveform-audio data. See the Platform SDK documentation for more information.

**WinMain**
: A function called by the system as the initial entry point for a Win32-based application. See the Platform SDK documentation for more information.

**ZeroMemory**
: A function that fills a block of memory with zeros. See the Platform SDK documentation for more information.

Previous Home Topic Contents Index Next

# Glossary

This glossary primarily defines terms related to Microsoft® DirectShow™. See the Microsoft Windows NT® or Microsoft Windows 95® online Help for additional information.

A B C D E F G H I *J* K L M N O P Q R S T U V W *X Y Z*

## A

**advise**
: A call that defines a point in time when the caller wants to be notified. See also periodic advise.

**application notification**
: An event that the controlling application retrieves from the filter graph manager.

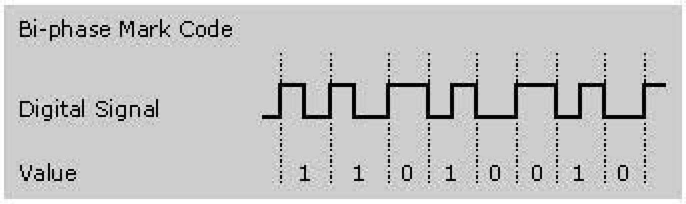
**asychronous reader filter**
: A file source filter that does no parsing on its own, but just reads data off the disk for playing back media files. This is the source filter used for most DirectShow™ filter graphs.

**ATR**
: Audio tape recorder.

## B

**backing object**
: In multimedia streaming, the backing object is either the DirectDraw surface for video or the IAudioData object for audio.

**bi-phase mark**
: A method of encoding digital data within an analog signal. Bi-phase mark code is self-clocking (signal alternates on entry and exit) and not polarity conscious (identical values can be on top or bottom). Because it can be read over a wide range of play speeds, it is used for LTC signals. (originally known as Manchester-1 code).



**blackburst**
A video signal that can be used to synchronize multiple video sources. Blackburst contains the vertical sync, horizontal sync, and the chroma burst information, but no active picture information.

**blit**
Bit block transfer. Used to transfer all or part of a bitmap from a source such as memory or the screen, to a destination such as another memory or display surface.

**bob**
A method of displaying interlaced video fields on a progressive monitor. See field for a more detailed description. See also weave.

Previous Home Topic Contents Index Next

# C

**chroma burst**
Provides color information about the active line of video to the color decoder.

**class factory**
A COM object that implements the **IClassFactory** interface and that creates one or more instances of an object identified by a given class identifier (CLSID).

**class factory template**
A template that contains information about a class that is vital to its framework. The DirectShow COM framework developers to provide a class factory template for each C++ class implementing a COM object. Class factory templates are defined using two global variables (g_Templates and g_cTemplates). For more information, see COM Objects in DirectShow.

**codec**
Compressor/decompressor. See compression filter.

**compression filter**
A specialized type of transform filter. Compression filters (compressors) accept data, use a compression scheme to transform the data, and pass the compressed data downstream.

**cutlist**
A collection of cutlist elements (clips). When you play a cutlist, it appears as if you're playing back one media file, but it is actually composed of clips from one or more media files. See About Cutlists for more information.

**cutlist element**
A piece (clip) from a media file. Cutlist elements make up a cutlist. See About Cutlists for more information.

## D

**debug build**
The largest and slowest type of build, but one that provides better facilities for debugging new code and tracking problems.

**debug level**
A DWORD value indicating the relative importance of a debug output, where zero is the most important level.

**downstream filter**
The next filter in line to receive data from an upstream filter. An upstream filter sends data from its output pin to the connected input pin of the downstream filter.

## E

**edit decision list**
Also known as an EDL. It is a list of items called events that describes a video sequence, audio sequence, or both, using SMPTE timecode to identify the video and audio source material. A sample event might look like this:

001 ABC123 V C 03:01:00:00 03:02:20:15 01:00:00:00 01:01:20:15

This means event 001 uses only video from reel ABC123, frames 03:01:00:00 up to but not including 03:02:20:15, and put it on a record tape as a straight cut starting at frame 01:00:00:00.

**effect filter**
A filter that applies an effect to media data, but doesn't change the media type.

**end-of-stream protocol**
A protocol that defines how filters generate and process end-of-stream information and how the filter graph manager is notified.

**error detection and reporting protocol**
A protocol that defines how errors are handled by filters and propagated to the filter graph manager.

**event notification**
A system-defined event sent from a filter to the filter graph manager. Filters pass these events to the filter graph manager by using the IMediaEventSink::Notify method, and

the application retrieves them with the IMediaEvent::GetEvent method. For more information, see Event Notification Codes.

Previous Home Topic Contents Index Next

## F

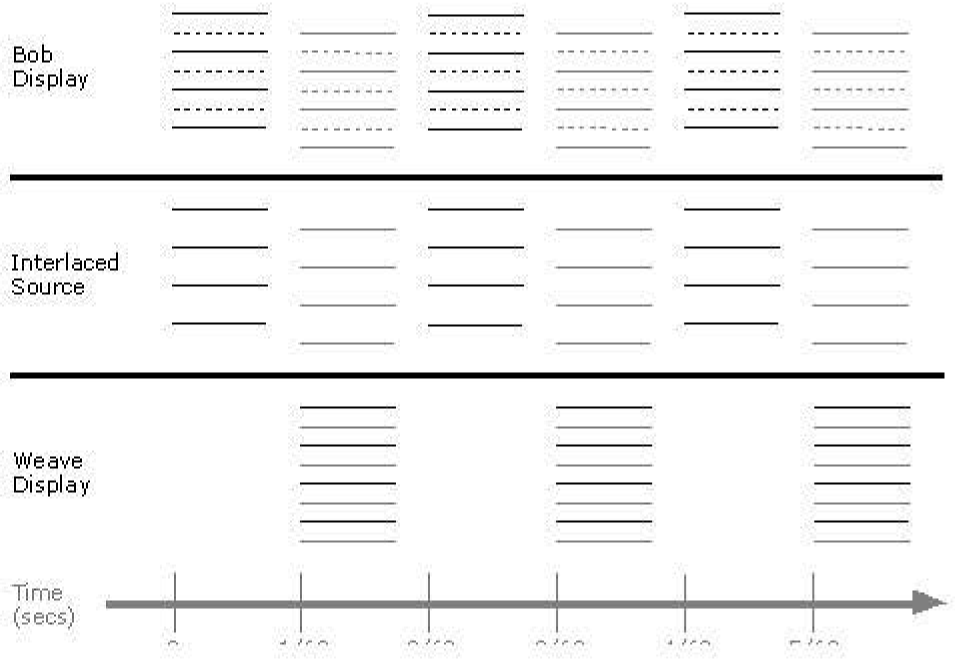
**field**

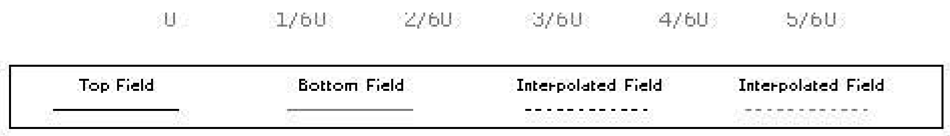
A portion of an interlaced video frame. NTSC provides a series of 59.94 interlaced fields per second, each separated by 1/59.94th of a second, with the scan lines of the even-numbered fields falling spatially halfway between the scan lines of the odd-numbered fields. No two fields are ever displayed on a television at the same time. You always look at either an even field or an odd field. Two interlaced video display methods are bob and weave.

Bob displays only one field at a time. In this mode, DirectDraw automatically shifts every other field by one half pixel vertically, and then stretches each field by a factor of two vertically.

Weave displays two successive fields at a time, where the resulting displayed image has horizontal scan lines that are alternately taken from the two fields. The fields are "woven" in a single frame. The top scan line of the displayed frame is the top scan line of the first field, the second displayed scan line is the top scan line of the second field, the third displayed scan line is the second scan line in the first field, and so forth.

The following diagram shows how fields are displayed in an interlaced source, using the bob and weave interlaced video display methods. In the diagram, top field refers to the odd, or first, field, and bottom field refers to the even, or second, field.





**file writer**
: The section of a filter graph that consists of the multiplexer and file writer filters.

**filter**
: A key component in the DirectShow architecture, a filter is a COM object that supports DirectShow interfaces or base classes. It might operate on streams of data in a variety of ways, such as reading, copying, modifying, or writing the data to a file. Sources, transform filters, and renderers are all particular types of filters. A filter contains pins that it uses to connect to other filters.

**filter graph**
: A collection of filters. Typically, a filter graph contains filters that are connected to perform a particular operation, such as playing back a media file, or capturing video from a VCR to the hard disk.

**Filter Graph Editor**
: A graphical tool included with the DirectShow SDK that creates and manages filter graphs. It enables you to easily create filter graphs by inserting filters, clicking, and dragging to form connections.

**filter graph manager**
: A component that oversees the connection of filters in a filter graph, and controls the media stream's data flow. Filters must be connected in the proper order, and the data stream must be started and stopped in the proper order. The filter graph manager does this, and can also search for a set of filters that will render a particular media type and build its filter graph. When an application starts, pauses, or stops the media stream, plays for a particular duration or seeks to a particular point in the stream, the filter graph manager calls the appropriate methods on the filters to implement this stream control.

**flushing protocol**
: A protocol that defines how filters flush data through the filter graph.

**format type**
: A GUID value that indicates what a format block contains. DirectShow defines a number of major types, for example, the video type. These major types have a format block, such as VIDEOINFOHEADER, that describes the media data. The format block for a particular media type is specified by a **GUID** in the AM_MEDIA_TYPE structure. This **GUID** is called the *format type*. If the format block contains **VIDEOINFOHEADER**, the format type **GUID** will be FORMAT_VideoInfo.

**frame blitting**
: Blitting (bit block transferring) media data to the client area of the application window on the primary surface.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

## G

**GUID**
A globally unique identifier used to uniquely identify objects, such as interfaces and plug-in distributors. Class identifiers (CLSIDs) and interface identifiers (IIDs) are **GUID**s. You can generate **GUID**s with the command-line utility program, UUIDGEN, provided with the Win32® SDK, or with the Microsoft Foundation Class Library (MFC) sample application, GUIDGEN, provided with Microsoft Visual C++®.

Previous Home Topic Contents Index Next

## H

**hue**
Color produced by visible light.

Previous Home Topic Contents Index Next

## I

**input pin**
A pin that accepts data into the filter.
**interlace**
Raster scan involving frames that are composed of two fields. See also bob and weave.
**interleaving**
The arrangement of corresponding video frames and audio samples in a file. Typically, one or more frames of video data are intermixed in the file with a certain number of samples of audio data. The data sequence is audio, followed by video, followed by audio, followed by video, and so on, rather than all audio data followed by all video data. The way a file is interleaved affects playback efficiency and the ability to stream the file.

Previous Home Topic Contents Index Next

## K

**kernel mode**
The processor mode that allows full, unprotected access to the system. A driver or thread

running in kernel mode has access to system memory and hardware.

**keyframe**
: A frame of video data that contains all the data necessary to construct that frame. In contrast, delta frames contain data relating to changes from the last keyframe and do not contain enough information by themselves to construct a complete frame.

Previous Home Topic Contents Index Next

# L

**letterbox**
: DVD-Video display format in which the video is sized as large as possible inside the display window, without any cropping or stretching. See DVD_PREFERRED_DISPLAY_MODE.

**Linear editing**
: A method of editing program material where the final product is built cut by cut in a sequential or linear fashion. Changes to any edit require re-recording all of the succeeding edits or using the recorded master as a source. These systems are found at the very low end and very high end of the editing equipment market. They are comprised of a control computer, video tape recorders and switchers, and generally do not provide random access disk-based storage of source material, although connection to a digital disk recorder will provide limited random access capability.

**LTC**
: Linear Timecode (formerly known as Longitudinal Timecode). Timecode is stored on a separate audio track and is one video frame time in duration.

Previous Home Topic Contents Index Next

# M

**major type**
: A GUID value that describes the overall class of media data for a data stream. Typical values are MEDIATYPE_Video, MEDIATYPE_Audio, MEDIATYPE_Text, and MEDIATYPE_Midi.

**merit**
: A value that controls the order in which the filter graph manager accesses your filter as a result of a call to IGraphBuilder::Connect, IGraphBuilder::Render, or IGraphBuilder::RenderFile. Possible merit values include MERIT_PREFERRED, MERIT_NORMAL, MERIT_UNLIKELY, and MERIT_DO_NOT_USE. See IFilterMapper::RegisterFilter for a description of merit values. When searching for a rendering configuration, the filter graph manager uses the filter mapper, which reads the registry and determines the types of filters available. The filter graph manager then

attempts to link filters that accept that data type until it reaches a rendering filter. A merit value is registered with each filter and, of the filters that might be capable of handling the data, the filters with the highest merit are tried first. The filter graph manager uses other criteria in the registration to choose between filters with equal merit. AMovieSetupRegisterFilter2 registers a filter's merit, pins, and media types in the registry by using the IFilterMapper2 interface.

**method**
: A predefined interface function.

**media sample protocol**
: A protocol that defines the way that media samples are allocated and passed between filters.

**media time**
: A term used to refer to positions within a seekable medium. Media time can be expressed in a variety of units, and indicates a position within the data in the file.

**minidriver**
: A hardware-specific DLL that uses a Microsoft-provided class driver to accomplish most actions through function calls and provides only device-specific controls. Under the Windows Driver Model (WDM), the minidriver registers each adapter with the class driver, which creates the device object. The minidriver uses the class driver's device object to make system calls.

**minor type**
: See subtype (media type).

**moniker**
: An object that implements the IMoniker interface. A moniker acts as a name that uniquely identifies a COM object. In the same way that a path identifies a file in the file system, a moniker identifies a COM object in the directory namespace.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

## N

**new segment protocol**
: A protocol that defines a way to present start and stop times and data rate information to the filter in advance of the data, so that a filter can adjust its processing accordingly.

**nonlinear editing**
: A method of editing program material where the final output product is not generated until all editorial decisions are made. All edit decisions are maintained in an edit list that drives a real-time preview. No recording is made until the edit decisions are finalized. This necessitates a real-time, multievent playback mechanism using digital video and audio stored on disk drives. These systems are used primarily for offline editing, although higher-end systems with broadcast-quality video are now available.

**NTSC**
: National TV Systems Committee analog composite color television standard. Uses a rate of 29.97 frames per second and 59.94 fields per second. North America, Japan, and many other countries use NTSC. See also PAL and SECAM.

## O

**object register**
The list of objects in a debug build that have been created but not yet destroyed in the CBaseObject class and classes derived from it.

**offline editing**
The process of using low-cost, nonoutput quality video and audio equipment to make editorial and creative decisions. This process results in an edit decision list describing the edits, and it is usually stored on a floppy disk or transferred to other systems over a network.

**online editing**
The process of using high-quality mastering equipment to produce a finished program. When fed by an edit decision list generated by an offline system, the process can be largely automated, although some high-budget programs bypass the offline process.

**output pin**
A pin that provides data to other filters.

## P

**PAL**
Analog composite color television standard, with a rate of 25 frames per second, and 50 fields per second. Much of Europe, Australia, and other parts of the world use PAL. See also NTSC and SECAM.

**pan-scan**
DVD-Video display format in which a 16 × 9 video is cropped for display in a 4 × 3 window, using parameters defined by the video author. See DVD_PREFERRED_DISPLAY_MODE.

**parser filter**
A filter that pulls information from a disk by using the asynchronous file reader filter, or from the Internet by using the URL moniker filter.

**performance build**
A build that adds performance measurement capabilities to a retail build.

**periodic advise**
A regular series of advise calls.

**pin**
A COM object created by the filter that represents a point of connection for a data stream on the filter. Pins provide interfaces to connect with other pins and transport data. Input

pins accept data into the filter, and output pins provide data to other filters. An input pin typically exposes the IPin and IMemInputPin interfaces. An output pin typically exposes the **IPin**, IMediaSeeking, and IQualityControl interfaces. A source filter provides one output pin for each stream of data in the file. A typical transform filter, such as a compression/decompression (codec) filter, provides one input pin and one output pin.

**plug-in distributor**
A COM object that exposes a particular control interface and implements it by calling the enumerator of the filter graph manager, finding which filters expose the control interface and communicating directly with those filters. The developer generally doesn't implement these interfaces directly.

**preroll**
The queuing of data in advance of the desired playback time or position. Preroll improves the accuracy of playback and recording. For example, the initial audio data within AVI files is often loaded in advance of the first video data to help synchronize the video and audio data. DirectShow filters often preroll data so they are ready to play immediately. Prerolling is also important in video editing and device control, because VCRs typically position the tape to a given "preroll" distance from the point at which to record or playback so that the recording or playback occurs cleanly.

**preroll time**
Preroll time is the time prior to the start position at which nonrandom access devices should start rolling. See preroll.

**presentation time**
The stream time at which the packets of data that a filter receives should be presented downstream or rendered. When a filter graph runs, each filter is passed a start time according to the reference clock, and the packets of data that a filter receives will usually be time-stamped with the presentation time.

**preview section**
The portion of the filter graph from the capture filter's preview pin downstream to and including the video renderer.

**primary surface**
The area in memory containing the image being displayed on the monitor. In Microsoft DirectX®, the primary surface is represented by the primary DirectDrawSurface object.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

# Q

**quality management protocol**
A protocol that defines how the filter graph adapts dynamically to hardware and network conditions to increase or decrease the media data flow. The IQualityControl interface is used to send quality control notifications from a renderer to an upstream filter or directly to a designated quality control manager, if one exists. (See quality sink.) The base classes implement passing quality control notifications upstream by providing the **IQualityControl** interface on the output pins of filters. Quality control notification uses a Quality structure, which indicates whether the renderer is overloaded or underloaded.

**quality sink**

A filter object designated by the IQualityControl::SetSink method to receive quality messages. When **IQualityControl::SetSink** is called, the filter is instructed not to send quality control messages upstream, but rather to send them to the object passed to the SetSink method. This object is called a quality-control manager. See quality management protocol.

## R

**reference clock**
An object that provides a clock and supports the scheduling of events according to time as counted by that clock. Any reference clock must support the IReferenceClock interface. The time of the clock can be obtained by calling the IReferenceClock::GetTime method. The time returned by **GetTime** is defined as a REFERENCE_TIME type and loosely represents the number of 100-nanosecond units that have elapsed since some fixed start time. The return value should generally increase at a rate of approximately one per 100 nanoseconds. In exceptional circumstances, the clock can stop for a time. (This will suspend any filter that was using the clock as a sync source.) The clock can also jump forward in exceptional circumstances.

**reference time**
An absolute time established by a reference clock in the filter graph. It refers to some time value outside the filter graph (for example, perhaps the time since Windows® was started). The reference time can be obtained by calling the IReferenceClock::GetTime method. The time returned is defined as a REFERENCE_TIME type and loosely represents the number of 100-nanosecond units that have elapsed since some fixed start time. Note that the reference time does not have to bear any permanent relationship to a real time. It can drift, it can drift at a changing rate, and it need not correct for such drift. In particular, it does not have to represent a count of the number of 100 nanoseconds that have passed since some arbitrary time in the past. See Understanding Time and Clocks in DirectShow for more information.

**renderer**
A filter that renders media data to any location that accepts media input. Most often, data is rendered to a computer monitor, sound card, or printer. Renderer filters have only input pins.

**rendering filter**
See renderer.

**retail build**
The smallest and fastest type of build, it is not as robust as a debug build.

**run time**
The control and a set of DLLs that enable you to play back the supported media types.

## S

**saturation**
: Color purity. For example, a color that is completely blue has a 100% saturation, while white, which is composed of all colors, has a zero saturation.

**SECAM**
: Analog composite color television standard, with a rate of 25 frames per second, and 50 fields per second. Much of Europe, Australia, and other parts of the world use SECAM. See also NTSC and PAL.

**seekable renderer**
: A renderer that reports EC_COMPLETE once when all seekable streams on that filter have reached the end of the stream. A seekable renderer is a renderer that supports the IMediaPosition object from the filter and has only input pins, or is a renderer whose input pins report that they are rendered. If the filters in a filter graph detect the end of the stream, the filters report it with the **EC_COMPLETE** event notification. The filter graph asks filters if they can report **EC_COMPLETE** through a seekable renderer.

**servo**
: The electromechanical system that maintains the proper speed and phase of a VCR's video head and tape transport.

**sink file**
: The current file into which media samples will be written.

**source filter**
: A filter that takes data from some source such as the hard drive, network, or the Internet, and introduces it into the filter graph.

**stream notification**
: An event that occurs in the media stream and is passed from one filter to the next.

**stream time**
: A time that represents the time the since the filter graph was last started. By definition, stream time is equivalent to reference time minus start time when the graph is running. Stream time is relevant only within a running filter graph. When a filter graph is run, each filter is passed a start time based on the reference clock, and the packets of data that a filter receives will usually be time-stamped with the stream time.

**subtype (media type)**
: A GUID value that describes the specific format of media data for a data stream. Typical values include MEDIASUBTYPE_MJPG, MEDIASUBTYPE_RGB8, MEDIASUBTYPE_RGB565, MEDIASUBTYPE_MPEGPacket, MEDIASUBTYPE_Avi, and MEDIASUBTYPE_WAVE. See AM_MEDIA_TYPE for more information.

**sync source**
: See reference clock.

Previous Home Topic Contents Index Next

## T

**tearing**

A visual rip effect that occurs when one displayed video frame contains parts of two different source video frames. It is usually caused by improper synchronization between video rendering and the graphics display. DirectShow uses double- or triple-buffered DirectDraw overlay surfaces when possible to prevent tearing.

In analog video terms, tearing occurs due to poor synchronization of a video signal at an edit point.

**timecode**

SMPTE timecode, more properly known as SMPTE time and control code, is a series of digital frame address values, flags, and additional data applied to a video or audio stream, and is defined in ANSI/SMPTE 12-1986. Its purpose is to provide a machine-readable address for video and audio.

**time stamp**

Time on a media sample indicating when it was recorded and when it should be scheduled for playback. Time stamps are measured in 100-nanosecond units (REFERENCE_TIME) and are normalized so that zero indicates when the graph is run. See Time Stamps for more information.

**transform filter**

A filter that takes data, processes it, and then passes it along to the next filter in the filter graph.

**transform-inplace filter**

A transform filter that can perform its operation in place (without copying data or altering the data's media type).

**transport**

The mechanism that channels audio data, video data, or both from an external device to the computer and from the computer to the external device.

**trimin**

The beginning time (starting position) of a cutlist element or clip within a media file. See About Cutlists for more information.

**trimout**

The ending time (position) of a cutlist element or clip within a media file. See About Cutlists for more information.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

## U

**upstream filter**

The filter that passes data from its output pin to the connected input pin of the next filter in the filter graph.

**URL moniker filter**

A source filter that reads from an Internet server.

**userbits**
: Undefined bits in the SMPTE timecode. These bits can be used for a variety of purposes such as identifying shot and take numbers, calendar date, client code, or any other information the user wants to encode so that it travels with the source material throughout the post-production process. Often, userbits contain original film and audio tape information. This information is used to generate negative cut lists, so a theatrical film can be edited electronically.

## V

**VITC**
: Vertical Interval Timecode. Timecode information stored in a video signal's vertical blanking interval. It is usually located on one or two lines somewhere between lines 10 and 20.

**vertical blanking interval**
: A synchronizing period in the video signal when no active picture information is transmitted.

**VPE**
: Video port extensions. Extensions to the DirectDraw API to control the video stream from the video port, within the context of VGA memory.

**VTR**
: Video tape recorder.

## W

**weave**
: A method of displaying interlaced video fields on a progressive monitor. See field for a more detailed description. See also bob.

**wildcards**
: Character that is substituted to represent multiple strings. For example, "?" represents any single character, "*" represents zero or more characters, and "#" represents any single digit (0-9). Thus, Strmbas?.lib would match both Strmbasd.lib and Strmbase.lib. Strm*.lib would match Strmbasd.lib, Strmbase.lib, and Strmiids.lib.

# Legal Information

[This is preliminary documentation and subject to change.]

## Broadcast Architecture Programmer's Reference

# About This Programmer's Reference

[This is preliminary documentation and subject to change.]

This November 1997 edition of the Broadcast Architecture Programmer's Reference presents documentation on developing software for *broadcast clients* and *broadcast servers*.

**Note** For information on developing hardware that supports Broadcast Architecture, see the Broadcast Architecture Device-Driver Kit (DDK). This DDK is part of the device-driver documentation for the Microsoft® Windows® 98 operating system, provided with the Windows 98 beta program.

The following sections provide more information about Broadcast Architecture and this reference documentation:

- Broadcast Architecture Defined
- Documentation Structure
- Prerequisites for Broadcast Architecture Development

# Broadcast Architecture Defined

[This is preliminary documentation and subject to change.]

The *Broadcast Architecture* documented here is an integral part of the Microsoft® Windows® 98 operating system. Broadcast Architecture enables personal computers to serve as *broadcast clients* for digital data networks and for analog broadcast networks.

Using Broadcast Architecture, video, audio, and computing data content for computer receivers can be broadcast over existing networks. Content can also be transmitted by analog cable or analog terrestrial means, and in later versions will be transmitted by satellite.

# Documentation Structure

[This is preliminary documentation and subject to change.]

The Broadcast Architecture Programmer's Reference contains the following sections:

- Overview, including these topics:
    - Introduction to Broadcast Architecture, which covers basic concepts of Broadcast Architecture.
    - How Broadcast Architecture Works, which briefly describes the hardware required, the software to be designed, and the software standards to follow for Broadcast Architecture.
- Broadcast Client, illustrating how to develop software for the broadcast client on the Microsoft® Windows® 98 operating system. This section includes these topics:
    - Broadcast Client Architecture, outlining the subsystems included in the broadcast client.
    - Client Hardware Requirements, describing hardware and operating system requirements of a broadcast client.
    - Television Services, describing the TV Viewer component, Television System Services, the Show References data format, and Show Reminders, which remind a user to watch or record a broadcast.
    - Program Guide Services, documenting services that support the *Program Guide* provided by Broadcast Architecture. This section presents an Overview of Program Guide Services, followed by sections on Updating Guide Data, Loader Libraries, Guide Data Objects, the Guide Database Schema, and Program Guide Registry Entries.
    - Streaming Video Services, documenting the streaming video services of Broadcast Architecture in the sections Video Control, Enhancement Video Control, and DirectShow Filter Reference.
    - Data Services, describing the broadcast data support provided by Announcement Listener, Video Enhancements and the Internet Channel Broadcast Client.
- Broadcast Server, covering development of broadcast server software. This section includes the following topics:
    - Broadcast Server Architecture, providing an overview of the server architecture used to broadcast data to clients in the home.
    - Writing Content Server Applications, discussing design and programming issues to address when creating a data service system for Broadcast Architecture.
    - Microsoft Multicast Router, documenting the *Microsoft Multicast Router* (MMR) which enables content servers to send data over a broadcast *head end*.
    - Internet Channel Broadcast Server, documenting the server supporting *Internet channel broadcasting* in Broadcast Architecture.
- Using Broadcast Architecture, providing information on how to develop software for Broadcast Architecture. This section includes the following topics:
    - Displaying Video, discussing how to present audio and video on a computer and how to manage devices associated with video.
    - Writing a Custom Guide Database Loader, covering how to write programs that load information into the *Guide database*.
    - Scheduling Show Reminders, describing how to schedule *show reminders* using your applications.
    - Creating TV Viewer Controls, discussing how to extend the Broadcast Architecture user interface by writing custom controls to be hosted by *TV Viewer*.

    - Writing Server Applications, covering how to write applications that send data to the MMR for routing to the broadcast network.
- Glossary, defining Broadcast Architecture terms.

# Prerequisites for Broadcast Architecture Development

[This is preliminary documentation and subject to change.]

The Broadcast Architecture Programmer's Reference assumes the reader is familiar with general concepts of programming for Microsoft® Windows® operating systems, such as:

- The *Network Driver Interface Specification* (NDIS) driver model
- The *Windows Sockets* (WinSock) application programming interface (API)
- The Microsoft® DirectShow™ API
- Stream class drivers based on the Windows Driver Model (WDM)

To locate more information on these or other subjects related to Broadcast Architecture development, see Finding Further Information. To locate sample applications that demonstrate concepts described in this programmer's reference, see Broadcast Architecture Sample Applications. For more information on working with type libraries in Broadcast Architecture, see Creating Type Libraries for Broadcast Architecture.

## Finding Further Information

[This is preliminary documentation and subject to change.]

Depending on what type of software you are developing, you may need to review information on programming topics.

You can obtain much of this information, including all documentation in the Platform Software Development Kit (SDK), through the Microsoft® Developer Network (MSDN). Microsoft strongly encourages you to enroll in Microsoft Developer Network Level 2 or greater. For enrollment information, contact your local reseller or call (800) 759-5474. For the most timely development information from MSDN, connect to the http://www.microsoft.com/msdn/ site.

The following lists indicate where to find documentation or other information on topics referred to in the Broadcast Architecture Programmer's Reference.

### Further General Information

- For information on NDIS, see the Microsoft® Windows NT® version 4.0 DDK. This DDK contains most NDIS information Broadcast Architecture developers require. Broadcast Architecture relies on NDIS version 5.0, a recent update; for any information specific to NDIS 5.0, see the NDIS 5.0 specification.
- For information on Windows base services, including the Microsoft® Win32® application

programming interface (API), see the Windows Base Services section of the Platform SDK. For information on the Win32 error function **GetLastError**, see the Debugging and Error Handling section within the Windows Base Services section.
- For information on WinSock version 2.0, see the Windows Sockets 2 section in the Platform SDK.
- For information on WDM stream class drivers and filters, including those that are part of the Windows 98 operating system, see the device-driver documentation for Windows 98. This documentation is provided as part of the Windows 98 beta program.
- For information on how to create show *enhancements* and other content data, see the *Design Kit for the Microsoft Broadcast Architecture* compact disc.
- For information on DirectShow, including information on constructing a custom filter graph and creating new DirectShow filters, see the DirectShow section in the Platform SDK.
- For information on video card requirements for Broadcast Architecture, see *Hardware Design Guide for Microsoft® Windows® 95*, available from Microsoft Press®.
- For information on the Component Object Model (COM) and on Microsoft® ActiveX™ technologies, see the COM and ActiveX Object Services section of the Platform SDK.
- For information on Data Access Objects (DAO), see the DAO SDK documentation supplied with the Microsoft® Visual C++® development system or with MSDN Level 2.
- For information on Structured Query Language (SQL), see the Microsoft® Jet Database SQL Reference in the DAO SDK documentation.
- For information on the Microsoft® NetShow™ server, see the Microsoft NetShow section of the Platform SDK, or the http://www.microsoft.com/netshow/ site.
- For information about the Task Scheduler feature of Microsoft® Windows® 98, see the Task Scheduler section of the Platform SDK.
- For information on CryptoAPI, see the CryptoAPI 2.0 section of the Platform SDK.
- For information on cryptographic service providers, see the Cryptographic Service Providers section of the Platform SDK.

## Further Information on Television Services for the Client

- For more information on using the Jet database, as required by some functions of *Television System Services* (TSS), see the DAO SDK documentation supplied with Visual C++ or with MSDN Level 2.
- For information on the **ITaskTrigger** interface and the **TASK_TRIGGER** structure used with TSS and with *show references*, see the Task Scheduler section of the Platform SDK.
- For information on the **IDispatch** COM interface used by the TSS object library and *TV Viewer* applications, see the Automation section of the Platform SDK.
- For information about using the Jet Installable Sequential Access Module (ISAM), see the DAO SDK documentation.
- For information on the **SAFEARRAY** and **VARIANT** data types, see the Automation section of the Platform SDK.
- For information about the **IEnumVARIANT** interface used in enumerating collections, see the Automation section of the Platform SDK.
- For information about **ITask** interface, see the Task Scheduler section of the Platform SDK.
- For information about the **GetActiveObject** function used when connecting to TV Viewer, see the Automation section of the Platform SDK.
- For information about the **IOleObject** interface and the **IOleObject::DoVerb** method used by the Broadcast Architecture **ITVControl** interface, see the COM section of the Platform SDK.
- For information about the idle processes used by **ITVControl**, see the **OnIdle** function topic in

Microsoft® Foundation Classes (MFC) documentation supplied with Visual C++ or with MSDN Level 2.

- For information on the **QueryDef** query definition, see the DAO SDK documentation supplied with Visual C++ or with MSDN Level 2.

## Further Information on Program Guide Services for the Client

- For information on accessing the Jet database engine from a Guide database loader when developing in Microsoft® Visual Basic®, see the Visual Basic documentation supplied with the product or with MSDN Level 2.
- For information on accessing a Jet database from a Guide database loader when developing in Visual C++, see the DAO SDK documentation supplied with Visual C++ or with MSDN Level 2.
- For information on the DAO objects **CdbDBEngine**, **CdbWorkspace**, and **CdbDatabase** and the DAO function **OpenRecordset** used with the Guide data object classes, see the DAO SDK documentation.
- For information on setting the path and file name of the Guide database, on adding or deleting Guide database users using DAO, and on running a DAO query on the Guide database, see the DAO SDK documentation.
- For information on specifying a SQL **Where** clause as part of setting user viewing restrictions, see the Microsoft Jet Database SQL Reference in the DAO SDK documentation.
- For information about the **SendMessageTimeout** and **PostMessage** functions used when performing event notification for Guide database loaders, see the Windowing section in the Setup and Systems Management Services section of the Platform SDK.
- For information about using the Jet Installable Sequential Access Module (ISAM) when loading show enhancements, see the DAO SDK documentation.

## Further Information on Streaming Video Services for the Client

- For information about the methods, properties, and events used with the Visual Basic **Extender** object when working with the Video control, see the Visual Basic documentation supplied with the product or with MSDN Level 2.
- For information on the Visual Basic Object Browser, see the Visual Basic documentation.
- For information about the **IFontDisp** interface used by the **Font** property of the **BPCVid** object for the Video control, see the COM section of the Platform SDK.
- For information on locale identifiers for use with Video control properties, see the International Features section in the Windows Base Services section of the Platform SDK.
- For more information about the **RGB** and **QBColor** methods of Visual Basic, see "Working with Color" in the Visual Basic documentation.
- For information on the **HWnd** handle property used with **BPCDevices**, see the User Interface Services section of the Platform SDK.
- For information on country codes for use with **CountryCode**, a property of the **BPCDeviceBase** object used with the Video control, see the DirectShow section of the Platform SDK.
- For information on the **BPCDeviceBase** property **Rate**, see the **IMediaPosition** object documentation in the DirectShow section of the Platform SDK.

## Further Information on Data Services for the Client

- For more information about services in Windows that support the *Session Description Protocol* (SDP), see the reference entry for the **ITSdp** interface in the TAPI Version 3.0 section of the Platform SDK.
- For information on the relationship between an announcement in Announcement Listener and a deletion notice for that announcement, see the *Session Announcement Protocol* (SAP) specification.
- For information on the **IPersistStream** interface used by Announcement Listener, see the COM section of the Platform SDK.
- For information on the stream compiler language and its syntax, see the stream compiler syntax reference included with the *Design Kit for the Microsoft Broadcast Architecture* compact disc.
- To find the stream compiler object library, Stream.dll, see the *Design Kit for the Microsoft Broadcast Architecture* compact disc.
- For information on File Transfer Service (FTS), a component of NetShow used to transmit enhancement files and their dependencies, see the Microsoft NetShow section of the Platform SDK.
- For information on the enhancement stream editor, see the *Design Kit for the Microsoft Broadcast Architecture* compact disc.
- For information on designing and creating enhancements for shows and on enhancement design issues, see the documentation included on the *Design Kit for the Microsoft Broadcast Architecture* compact disc.
- For more information on the CSS positioning feature of Dynamic Hypertext Markup Language (DHTML) for use in enhancement layouts, see the Dynamic HTML section of the Internet Client SDK or the Internet Explorer Reference section of the Platform SDK.
- For information on the channel subscription mechanism of Microsoft® Internet Explorer version 4.0, used in Internet channel broadcasting, see the Internet Client SDK or the Internet Explorer Reference section of the Platform SDK. This documentation includes information on channels and *Channel Definition Format* (CDF) files.
- For additional information on Internet Explorer 4.0, see the http://www.microsoft.com/ie/ site.
- For the CDF specification and related information, see the http://www.microsoft.com/standards/cdf-f.htm site. The site http://www.microsoft.com/workshop/prog/ie4/channels/cdf1-f.htm also provides information about channels and CDF.
- For more information on Active Server Pages for use in Internet channel broadcasting development, see the Active Server Pages section of the Platform SDK.

## Further Server Information

- For documentation on the WinSock structure **FLOWSPEC** used with *Microsoft Broadcast Data Network* (MSBDN) functions, see the Windows Sockets 2 section of the Platform SDK.
- For information on the speed of the serial interface to use for communication with the TES-3 encoder, information used by the serial VBI output driver, see the documentation for the Norpak TES-3 encoder.
- For documentation on the Win32 API functions that output driver functions call, see the Windows Base Services section of the Platform SDK.
- For documentation on the Win32 API function **GetTempPath**, used in setting system options for an Internet Channel Broadcast server, see the Files and I/O section in the Windows Base Services section of the Platform SDK.
- For information on the channel subscription mechanism of Internet Explorer 4.0, used in

Internet channel broadcasting, see the Internet Client SDK or the Internet Explorer Reference section of the Platform SDK. This documentation includes information on channels and CDF files.

- For additional information on Internet Explorer 4.0, see the http://www.microsoft.com/ie/ site.
- For the CDF specification and related information, see the http://www.microsoft.com/standards/cdf-f.htm site. The site http://www.microsoft.com/workshop/prog/ie4/channels/cdf1-f.htm also provides information about channels and CDF.

### Further Information on Development Tasks in Broadcast Architecture

- For more information on working with display surfaces, also called drawing surfaces, see the documentation for the Microsoft® DirectDraw® API in the DirectX 5 section of the Platform SDK.
- For information on the **CdbDBEngine** object used when creating a custom database loader, see the DAO SDK documentation supplied with Visual C++ or with MSDN Level 2.
- For information about the **GetActiveObject** function used when connecting to TV Viewer, see the Automation section of the Platform SDK.

## Broadcast Architecture Sample Applications

[This is preliminary documentation and subject to change.]

A group of sample applications, controls, and libraries for use with Broadcast Architecture is provided with the Platform SDK and with the Windows 98 beta program.

In the Platform SDK, text files including sources, libraries, and control files needed to create executable versions of the samples reside in the *SDKDIR*\Samples\Graphics\Ba directory. Note that *SDKDIR* in this path represents either the Mssdk directory on the Platform SDK compact disc, or the directory you selected for installing the Platform SDK.

If you are a member of the Windows 98 beta program, you can also get these samples by downloading the file Basamp.zip from the World Wide Web page at http://winbeta.microsoft.com/bpc/.

The samples are as follows:

- Four samples developed in different ways demonstrate how to use the Video control to display video. These samples are located in the UseVideo subdirectory:
  - The Vid_Tune subdirectory contains a Visual Basic group comprising a Visual Basic control and a Visual Basic project to test the control. The Visual Basic control contains a Video control as a constituent control.
  - The VidCntrl subdirectory contains a Visual Basic project that uses the Video control.
  - The VideoMFC subdirectory contains a Visual C++ project workspace that uses the Video control.
  - The WebTune subdirectory contains a Hypertext Markup Language (HTML) document that Internet Explorer uses to display a Video control.

For more information on how to run each sample, see the Readme.txt file in the directory for that sample.

- Two samples developed in different ways, Load and Download, demonstrate how to write Guide database loaders. The Loader subdirectory contains source code for the Load sample, a simplified loader that gives an unobstructed view of how certain loader functions work. The Download subdirectory contains source code for the Download sample, which shows how to write a full-featured loader. For more information on each sample, see the Readme.txt file in the subdirectory for that sample.
- The sample application Schsamp.dll is an ActiveX component that schedules a show reminder. It is located in the subdirectory Schsamp. For information on how to compile and run Schsamp.dll, see the Readme.txt file in that subdirectory.
- The sample MFC application Tvxsamp.exe demonstrates how to connect to and control TV Viewer. It is located in the subdirectory Tvxsamp. For information on how to compile and run Tvxsamp.exe, see the Readme.txt file in that subdirectory.
- The sample applications Wsend and Wlisten send and receive multicast data. In addition, Wsend sends data to the Microsoft Multicast Router using a tunnel. The sample library Brtest includes functions used by Wsend and Wlisten. These samples are located in the subdirectories Brtest, Wsend, and Wlisten. For more information on how to compile and run each sample, see the Readme.txt file in the subdirectory for that sample.

## Creating Type Libraries for Broadcast Architecture

[This is preliminary documentation and subject to change.]

Broadcast Architecture does not provide separate type library files for its components. Instead, type library information is included as a resource in the .dll or .ocx file for the component in question. Including type library information as a resource ensures that this information is always the correct version for the object library. To create a type library from a .dll or .ocx file, use the type library utilities provided by your compiler.

If you are using Visual C++, the type library utility is the ClassWizard available from the **View** menu. In Microsoft® Visual J++™, use the Java Type Library Wizard available from the **Tools** menu. In Visual Basic, simply create a reference to the object library or add the component to the project.

# Introduction to Broadcast Architecture

[This is preliminary documentation and subject to change.]

Broadcast Architecture enables personal computers to receive video and digital data from virtually any broadcast source, including existing satellite, cable, and terrestrial television networks. Effective with the Microsoft® Windows® 98 operating system, every copy of Windows shipped includes extensions that support Broadcast Architecture. Any computer running these extensions, and equipped with enabling hardware, can receive interactive entertainment and information delivered through high-speed, nationwide broadcast channels.

Broadcast Architecture offers an evolutionary step in the progress of computers and television. By using Broadcast Architecture, you can build upon your existing content and technologies, while creating added value by combining them in new ways. At the same time, Broadcast Architecture suggests new avenues for the distribution of home entertainment, news, information, education, software, and general retail merchandise.

The following sections introduce the concepts, goals, and implications of Broadcast Architecture:

- Concept: Computers Receive Broadcasts. In a nutshell, Broadcast Architecture allows personal computers to receive broadcasts transmitted over any kind of network.
- The Broadcast Push Model. Why broadcasting is an efficient way to deliver large amounts of information.
- Computer Power. As a receiver of television, radio, and data broadcasts, a personal computer has many advantages over older technologies.
- New Kinds of Television. The combination of television and computers creates a new medium that not only provides information and entertainment for users, but also offers many new opportunities for advertisers, broadcasters, network operators, all types of publishers, and hardware and software manufacturers.
- New Kinds of Data. New types of content can be designed and created specifically for *broadcast clients*. At the same time, broadcast client viewers can watch standard television with top-of-the-line picture quality and sound.

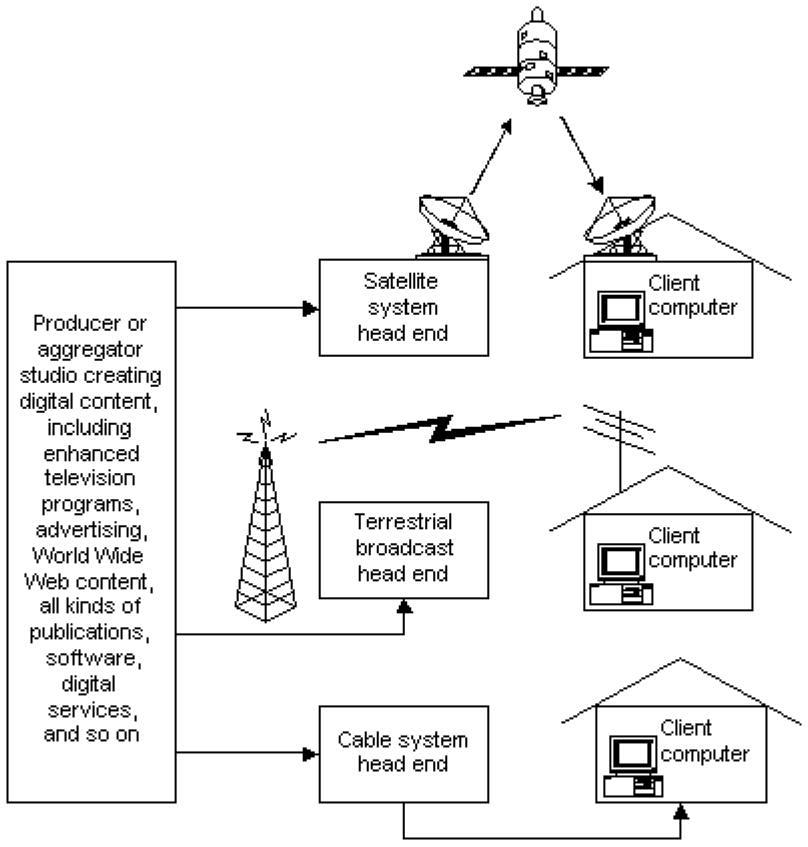
# Concept: Computers Receive Broadcasts

[This is preliminary documentation and subject to change.]

Broadcast Architecture relies on *Internet Protocol* (IP) *multicast* and other standard technologies to let a personal computer receive unidirectional digital and analog transmissions over *any* kind of television or computer network. Broadcast Architecture is transport independent; in other words, Broadcast Architecture data can travel over any standard data-transfer system.

Following a standard client/server computing model, broadcast-enhanced computers are equipped to act as "clients." In other words, they serve as data tuners that receive and process streams transmitted to them by broadcast "servers."

The following diagram illustrates client computers receiving broadcast data over different kinds of television network.


Producer or aggregator studio creating digital content, including enhanced television programs, advertising, World Wide Web content, all kinds of publications, software, digital services, and so on
Satellite system head end
Client computer
Terrestrial broadcast head end
Client computer
Cable system head end
Client computer

A broadcast client can easily be equipped with appropriate receiver cards and supporting software to receive broadcasts in virtually any format from almost any source. Whether the transmission travels by satellite, cable, conventional terrestrial antenna, or over a computer network, whether the signal is digital or analog, whether it is a video, audio, or binary data stream, broadcast clients are designed to accommodate it. Clients accommodate all these types of data in a standard way using the Microsoft® Windows® 98 operating system. In fact, the same broadcast client can receive almost any combination of sources and data types.

Almost all the technology and infrastructure needed to create broadcast clients is currently in place. In particular, broadcast clients rely on broadcast networks and a phone-line *back channel* that are inexpensively and reliably available in virtually every home today.

Broadcast clients provide a particularly flexible and cost-effective path to television of the future. Not only do they have a lot to offer immediately, but also they support low-cost incremental steps to higher television resolutions, growing back-channel bandwidth, increasing interactivity, and new multimedia forms of television. For viewers and content producers alike, broadcast clients provide painless interim solutions at every stage of the path. Rather than becoming obsolete when new technology becomes available, they are designed to incorporate technological advances smoothly.

Broadcast client software has been designed so that some simple, impressive combinations of television and information content can immediately be delivered in the form of World Wide Web pages. Broadcasters can thus take advantage of standard Web design tools, scripts built in the Microsoft® Visual Basic® programming system, and skills they already have to create multimedia television rapidly and easily. In the cases where Web functionality is insufficient, programmers can readily take advantage of the power of the operating-system software provided for the broadcast client to write special-purpose applications.

## The Broadcast Push Model

[This is preliminary documentation and subject to change.]

Data transmitted over a computer or broadcast network using a unidirectional "push" model can reach very large numbers of people much more efficiently than the bidirectional "pull" model used on most computer networks today.

Computer networks today, including the Internet, generally use a pull model for transmitting data. In a pull model, a client sends a request for specific data to a server. The server processes the request and then sends back the requested information. In this model, clients "pull" information from the server. The process breaks down when too many clients request information at once. This breakdown occurs because even very powerful servers can handle only a limited number of requests at a time, and they must send a separate response to every request.

In a true push model, by contrast, the server broadcasts a large amount of information onto the network on its own schedule, without waiting for requests. The clients scan incoming information and save the parts users have indicated interest in, while discarding the rest. In this model, one server transmission can service an unlimited number of clients at once. In cases where many people need the same information, this push model is a much more efficient use of network bandwidth than the pull model.

The combination of a true push model and the very high bandwidths of existing broadcast channels allows Broadcast Architecture to deliver large quantities of data to customers conveniently. Such data can include video, audio, high-resolution images, large aggregated blocks of World Wide Web pages, databases, software, and data in other formats. This kind of data is generally too large to send or receive over telephone connections on a regular basis, even with the fastest modems. Broadcast Architecture can deliver this data to a client automatically, in the background, without the customer ever having to dial in, tune in, or download anything.

Soon, satellite broadcast digital data streams with capacities of between 1.2 and 6 megabits per second will be available on broadcast clients. At 1.2 megabits per second, a channel can transfer over 10 gigabytes every 24 hours, while 6 megabits per second translates into more than 60 gigabytes of information per day. By contrast, a compact disc today holds about two-thirds of a gigabyte. In future versions of Broadcast Architecture, a variety of 30-megabit-per-second channels may each deliver over 300 gigabytes per day.

Not only is broadcasting an efficient way to distribute information, existing broadcast networks already reach an enormous national and international audience. *Digital Household Report* of August 31, 1996, projected that 96.9 million households in the United States would receive analog and digital broadcast transmissions by the end of 1996. Broadcasts also reach a growing number of households internationally as broadcast satellite networks continue to proliferate.

Broadcast Architecture uses the existing standard *Internet Protocol* (IP) for broadcasting data. Over the Internet and other computer networks that make use of IP, broadcasts take the form of *IP multicasts* sent to many recipients at once, in contrast to usual *unicasts*, which are directed to a single recipient. In a corporate context, multicasting can greatly reduce network traffic over intranets when compared to unicasting the same data to the same recipients. A broadcast client is a perfect client for corporate multicasts both because of its high-bandwidth capabilities and because Broadcast Architecture handles *all* broadcast data as standard IP multicasts.

## Computer Power

[This is preliminary documentation and subject to change.]

Broadcast Architecture applies the versatility and power of computers in home or office to provide capabilities far beyond what televisions equipped with set-top boxes can achieve. Even before new programming becomes available, broadcast clients provide an appealing integration of excellent picture and sound quality with a state-of-the-art multimedia computer.

The computer gives viewers a great deal of flexibility in planning television programming, and it displays *Program Guide* information very conveniently and clearly. Using a modem connection to the Internet, data services of all kinds are available today.

### State-of-the-Art Computer Capabilities

To start with, you have all the power of a state-of-the-art computer, including multimedia and game playing. New power-saving instant response technology keeps the system available at all times. The Internet is fully accessible by modem, and some Internet content is already coordinated with television, as in the case of the MSNBC news network.

### Digital Display Technology for Television Signals

In addition, broadcast clients offer an incremental, flexible, and affordable migration path to higher resolution television. Instead of buying a whole new, very expensive digital set every time broadcast technology shifts to a new standard, all users have to do is plug a new card into their computers.

### Powerful Control over Programming Choices

Broadcast clients display program guide information of many different sorts in a single form. In this combined program guide, shows can easily be previewed, scheduled, and if appropriate paid for.

Broadcast clients offer the convenience and ease of use of finding, selecting, and scheduling entertainment and information in one place, using one familiar interface.

# New Kinds of Television

[This is preliminary documentation and subject to change.]

Broadcast Architecture transforms television into a multimedia experience. The combination of television with broadcast digital data displayed on a computer offers a new world of entertainment possibilities. It gives you the flexibility to become more actively involved in the television programs you watch, when you want to.

Digital data broadcasts synchronized to television shows can provide all kinds of annotation and extension of existing television and advertising content, using *Hypertext Markup Language* (HTML), Microsoft® Visual Basic® scripts, and controls based on the Microsoft® ActiveX™ technology platform.

The combination of television and computer content can offer these new possibilities, among many others:

- During sporting events, you can ask about statistics, track your favorite players or teams, and scan your preferred syndicated commentary.
- During dramas and comedies, you can locate cast information, recaps of past episodes, links to related Internet and bulletin board sites, and other such background information.
- Educational programs can supplement their presentations with broader and deeper information than fits into their time slots, together with links and references to additional resources.
- News and weather reports can be accompanied by local or other specialized information that satisfies the needs of limited audiences.
- Music-only channels can add background graphics containing song title, album, and artist information, so you know what you are listening to and how to find it again.

With the addition of a *back channel*, viewers can interact not only with the computer but also more directly with broadcasters, advertisers, and other viewers. A secure back channel also offers an unprecedented opportunity to sell directly into people's homes, letting them purchase from the comfort of their couches.

**Broadcast Architecture Provides Potential for Immediate Viewer Response**

Even without a back channel, television and digital data broadcasts can be synchronized and combined to let viewers interact on-screen with shows so as to play games, obtain supplementary information, test their knowledge or skills, and so on. *Content providers* can use common tools for World Wide Web site design to create enhancements for their shows, delivered as HTML pages.

By taking advantage of the low-cost, low-bandwidth back channel offered by the computer's modem, however, advertisers can also actually solicit real-time responses from viewers. Viewers can, for example, express product preferences, or they can inquire about products of interest.

Using the back channel, viewers can also vote on issues presented in a show, express opinions, take part in polls, play along with game shows, enter contests, and take quizzes. In addition, consumers can use the back channel to purchase products from the comfort of their living rooms.

## New Kinds of Data

[This is preliminary documentation and subject to change.]

Broadcast channels provide a fast and inexpensive way of distributing information. The broadcast client can monitor digital data streams 24 hours a day to keep caches of subscribed information up-to-date, without using the phone.

Broadcast clients are designed to filter high-bandwidth broadcast data streams and save whatever the viewer may have subscribed to, requested, or purchased. In combination with the broadcast client's system security and strong encryption, this capacity provides a reliable and economical channel for selling even high-priced or confidential digital goods and services.

Digital data can not only be sent over digital networks, it can also be incorporated into analog television signals in the *vertical blanking interval* (VBI). Digital data automatically transmitted in the VBI includes the *Program Guide* data provided by StarSight and other companies. Digital transmissions can also use analog bandwidth directly through *broadband* modems, such as cable modems.

Digital data broadcasts are ideal for delivering such time-sensitive information as stock prices, local news and weather, product catalogs, software updates, and much other information provided by subscription services. For example, frequently visited Internet sites can easily be broadcast in this fashion and cached on a hard drive. They are thus instantly available in an up-to-date form when a viewer wants them, perhaps in conjunction with an associated show or advertisement. Caching such sites eliminates frustrating busy signals, slow server response, and long download times that may be associated with visiting the sites using a modem.

Not only can broadcasting and caching popular Internet content make visiting useful pages much less frustrating for consumers, it can also reduce server loads so that other interactions, such as purchases, can proceed without delay.

In addition, background images, video, and music downloaded over high-bandwidth broadcast channels can be used to make computer experiences more attractive and fun, particularly in the realm of entertainment software and educational programs. All kinds of digital information previously difficult or very time-consuming to acquire can be made immediately available. For example, games or courseware might be regularly updated with new scenarios, information, backgrounds, and so forth to provide users a constantly changing landscape of interactivity.

# How Broadcast Architecture Works

[This is preliminary documentation and subject to change.]

Broadcast Architecture capabilities are achieved through a combination of hardware and software components that allows personal computers to serve as *clients* of *broadband* digital and analog broadcast networks. The following sections provide more information on these components:

- Overview of the Hardware
- Overview of the Software
- Sticking to Standard Technologies
- System Software Components
- System Software Extensions

## Overview of the Hardware

[This is preliminary documentation and subject to change.]

To achieve their purposes, broadcast clients require hardware components of the following sort:

- A digital-ready display. Broadcast clients are intended for home display of television. To achieve this goal requires a progressive-scan super VGA (SVGA) monitor with a resolution of at least 800 x 600 pixels, a refresh rate of 60 or 120 hertz, and phosphors with matching persistence so as to minimize flicker while matching the luminance of television picture tubes. Suitable monitors vary in size from under 17 inches to over 35 inches.
- A network receiver card. The receiver card, installed on a PCI or other high-bandwidth bus, provides functionality such as data tuning, decryption, demultiplexing, and other capabilities that permit it to receive signals from specific broadcast sources.
- A video card capable of *MPEG-2* compression. A video card with MPEG-1 and MPEG-2 audio and video decoding capabilities, also installed on a PCI or other high-bandwidth bus, must provide functionality such as SVGA video display, MPEG decoding, and *NTSC* or *PAL* signal demodulation and encoding. This functionality is needed for digital video disks (DVDs) as well as for videotape input/output and ordinary television broadcasts.
- A standard modem. Customers require a *back channel* to purchase pay-per-view movies and other premium data services and to interact with advertisers and broadcasters. A 14.4-kilobit-per-second or faster telephone modem provides a low-bandwidth back channel that is more than adequate for these needs. As *Integrated Services Digital Network* (ISDN), *Asymmetrical Digital Subscriber Line* (ADSL), and cable modems become widely available, faster back channels will become practical. As they do, more dynamic forms of interactive entertainment will evolve.
- A keyboard, pointing device, and remote control. Viewers must be able to control a broadcast client, together with other consumer electronic devices such as a VCR or stereo system,

comfortably and conveniently from a seat some distance from the screen. The keyboard, pointer, or remote control device or devices that meet this need should be wireless and designed for easy use on the lap.
- Custom hardware device drivers. Vendors of special-purpose cards to be part of the system need to provide standard device drivers for their hardware based on the *Network Driver Interface Specification* (NDIS).

Detailed minimum and recommended hardware configurations are specified in Client Hardware Requirements. However, within the given boundaries, alternate configurations are encouraged, as long as good television and data reception for broadcasts continues to be provided.

## Overview of the Software

[This is preliminary documentation and subject to change.]

Broadcast clients must be able to handle a wide variety of very high-bandwidth streams. These streams require different kinds of special-purpose hardware for reception and processing. The software that makes this demanding environment work is characterized by three design objectives:

- To use industry-standard technology wherever possible.
- To expose and document extensions specific to Broadcast Architecture.
- To maintain adequate security to protect all parties using Broadcast Architecture.

The software needed to make a broadcast client work can be divided into three categories:

- Operating-system software. The system software provided with Broadcast Architecture is based on industry standards to ensure its reliability and ongoing development.
- The container for Broadcast Architecture applications. Broadcast Architecture applications all run within the standard browser container provided by Microsoft® Internet Explorer. This container allows new applications downloaded by data broadcasters to take full advantage of all broadcast client capabilities, just as if the applications were integral parts of the system.
- Broadcast Architecture applications. These applications can be controls or scripts associated with World Wide Web pages, any other Internet application supported by Internet Explorer, or even ordinary Windows-based programs running outside the Internet Explorer container. Among the most important Broadcast Architecture applications is the *Program Guide* control that lets viewers search, sort, filter, select, and schedule television shows and other content of all kinds.

To locate documentation on creating specific types of Broadcast Architecture applications, see Documentation Structure.

# Sticking to Standard Technologies

[This is preliminary documentation and subject to change.]

Wherever possible, Broadcast Architecture software relies on standard solutions that are widely accepted, understood, and supported in the industry. These standards include:

- *The Transmission Control Protocol/Internet Protocol (TCP/IP) networking protocol*. This protocol is the one used by the Internet. By using TCP/IP as their primary networking protocol, broadcast clients make a standard way available to communicate with virtually any network in the world.
- *MPEG-2 compression*. This format is becoming the most widely accepted standard for delivering compressed video and audio and related data.
- *The Microsoft® Windows® 98 operating system*. In addition to being the successor to the most widely used and understood 32-bit operating system, Microsoft® Windows® 95, Windows 98 includes a number of components that are becoming or have become standards themselves:
  - Windows Sockets (WinSock) version 2.0. This application programming interface (API) provides a network abstraction layer that allows applications to receive and send network data without needing any information about the network involved. WinSock also provides access to TCP/IP.
  - *Network Driver Interface Specification* (NDIS) version 5.0 ports. The NDIS standard allows hardware device drivers to be written independently of the target operating system.
  - CryptoAPI. This API provides an abstraction layer for encryption and decryption services, so that applications can use different encryption methods without requiring any information about the hardware or software involved.
  - Microsoft® Internet Explorer. By incorporating Internet Explorer technology, the broadcast client can take advantage of all the latest Internet and Web enhancements.
  - Component Object Model (COM). This open standard allows different software modules, written without information about each other, to work together as if they were part of the same program.
  - *Microsoft DirectShow* (formerly called Microsoft® ActiveMovie™). This Microsoft® ActiveX™ technology provides an extremely flexible and capable architecture for managing and playing interrelated multimedia streams, which the broadcast client relies on. The key concept of the Microsoft® DirectShow™ API is to connect many independent *filter* programs together. Each filter handles a part of the process of receiving, decoding, transforming, scheduling, and displaying interdependent video, audio, and data streams.
  - Key codes for television remote controls. The remote control buttons included on keyboards and other devices communicate with broadcast clients by using standard Windows key codes. Use of standard key codes makes integration of remote functions into hardware very simple for manufacturers.

The DirectShow technology and the related stream class driver technology, part of the *Windows Driver Model* (WDM), is sufficiently important for broadcast clients that its flexibility should be stressed. DirectShow filters are modular software components that work together to process a data stream. When one feature of the stream changes, only the filter dealing with that feature need be replaced. For example, if a stream's encoding changes, only its decoding filter is affected. This modularity makes it easy to use and support clients that work with virtually any kind of broadcast possible.

To get the maximum performance, Broadcast Architecture supports DirectShow by using WDM stream class drivers. These drivers operate on data in *kernel mode*. DirectShow provides control of these drivers to applications through the use of proxy filters running in *user mode*. For example, an application can call a proxy filter in DirectShow to change channels on a *television tuner card*. Then the proxy filter calls the WDM stream class driver, which controls the television tuner card.

To locate more information on how to create and use new DirectShow filters to handle changing technologies, and on WDM stream class drivers and filters, see Further General Information.

## System Software Components

[This is preliminary documentation and subject to change.]

Operating-system software components particularly important to software development for Broadcast Architecture include:

- The *Program Guide* control, which gives a user access to the *Guide database*.
- The *TV Viewer*, a Microsoft® ActiveX™ container. TV Viewer is the primary user interface of the Broadcast Architecture component in the Microsoft® Windows® 98 operating system.
- The standard Broadcast Architecture control that parses and displays *Hypertext Markup Language* (HTML)

Of these, probably the most visible one is the Program Guide control. In clear graphical form, the Program Guide control displays information on all broadcast programming available to the user over various time periods. With this control, the user can search for favorite shows, keep track of episodes, set up reminders to watch or record shows, and watch previews. This control is continuously updated with Program Guide information from various providers.

The primary user interface of the Broadcast Architecture system component, TV Viewer, is an ActiveX container that hosts several important types of control. One type of control can play full-motion video over some or all of the screen. In other words, ActiveX controls within the TV Viewer container create the television video and audio that viewers see and hear. Another type of control displayed in TV Viewer uses Microsoft® Internet Explorer technology to interpret and display Web pages, either in conjunction with video or on their own.

This control, the standard Broadcast Architecture control that parses and displays HTML data, can also process scripts and host any ActiveX controls that a broadcaster chooses to include with a show. Because HTML is the language of the World Wide Web, the presence of this control not only means that a large amount of Internet data can immediately be combined with television programming, but also that there are existing, good tools for quickly designing and creating new HTML content without software development. A broadcaster can use triggers to synchronize the display of HTML material with television and can add, update, or remove HTML material without any intervention on the viewer's part.

Furthermore, through scripts and ActiveX controls included with HTML pages, broadcasters and independent software vendors can take full advantage of the broadcast client's computing capabilities to enhance a show in new ways, to provide complex interactivity with viewers, and to deliver valuable services. To locate more information on doing so, see Further General Information.

Although Broadcast Architecture software is designed to be open to developers, great care has been taken to ensure that system security can be maintained. Broadcast clients can thus provide a secure platform for commercial transactions.

## System Software Extensions

[This is preliminary documentation and subject to change.]

The operating-system software that forms part of Broadcast Architecture is intentionally based on industry standards, as described in Sticking to Standard Technologies. These standards guarantee developers, manufacturers, and viewers a stable, well-supported platform that will gracefully evolve to support new technologies as they appear.

In some areas, extensions to these standard technologies have been required. These extensions include:

- Extensions to accommodate the very large bandwidth occupied by high-quality digital audio and video streams, and to provide for flexible control over these streams. These extensions are special filters based on the Microsoft® DirectShow™ application programming interface (API). To locate more information on DirectShow, see Further General Information.
- Extensions to support interactions with users. Broadcast clients use specialized software to control, process, and display high-bandwidth broadcast streams. Some of this software, such as the *Guide database* system, also handles user interactions. The Guide database provides a single secure repository for program data. At the same time, any *service provider* can deliver application data to the database by writing a loader program.
- Extensions to support Television System Services (TSS), which allows users to tune to, schedule, and control available broadcasts. For more information, see Television System Services.

Microsoft also provides other types of system software extensions, such as a transport layer specifically designed for broadcast needs. Because such components all function as part of Microsoft® Windows® 98, third-party programmers can easily write programs that take advantage of them.

Microsoft makes interfaces that support operating-system extensions available through the Platform Software Development Kit (SDK), the Device-Driver Kit (DDK) for Windows 98, and the Web site provided by the Windows 98 beta program. The Broadcast Architecture Programmer's Reference and the Broadcast Architecture DDK document these interfaces.

# Broadcast Client Architecture

[This is preliminary documentation and subject to change.]

The architecture of the broadcast client enables computers based on the Microsoft® Windows® 98 operating system to be client systems for many types of broadcast network. The *broadcast client* is a standard computer enhanced with hardware and software components that enable it to process video, audio, and data from a variety of sources. Some networks that broadcast clients can support include:

- The Internet's multicast backbone (MBONE), and the networks and telephone lines that support it — Integrated Services Digital Network (ISDN), Ethernet, Asymmetrical Digital Subscriber Line (ADSL), and others.
- Cable television networks, such as Comcast and Tele-Communications, Inc. (TCI)
- Wireless cable networks.
- Conventional television networks, and television networks that broadcast digital data to viewers in the *vertical blanking interval* (VBI) of the television signal, such as Intel Intercast.

Some networks used by Broadcast Architecture have several unique aspects that must be considered because they are one-way data streams. These networks cannot run server applications that require a *back channel* to receive data from the client. The architecture of the broadcast client allows the use of another network, such as the Internet, to provide this back channel. Another effect of the one-way nature of broadcast networks is that the client cannot request that a bad packet be resent. To reduce this problem, the server must periodically resend data so the client can replace any corrupted or lost data. Finally, broadcast networks are capable of very high data transfer rates. Devices for broadcast networks, and the software associated with these devices, must be capable of handling this data rate without burdening the CPU.

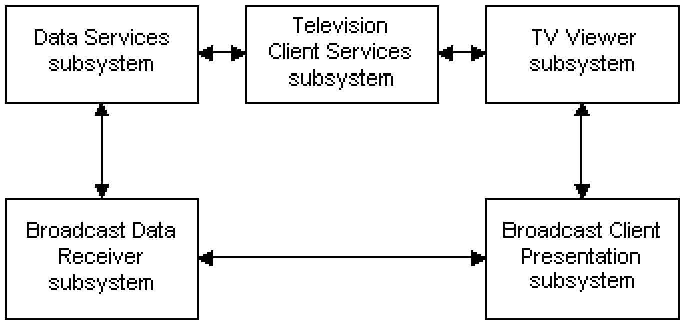
The high bandwidth of many broadcast networks make them ideal for transmission of audio and video data as well as computing data. Broadcast Architecture uses a set of modular components to receive, process, and present this data. This modular design means application developers can create applications without regard for the underlying technology. Hardware vendors can provide drivers to make their devices compatible with other broadcast client components.

# Broadcast Architecture Subsystems Overview

[This is preliminary documentation and subject to change.]

The broadcast client consists of a number of functional subsystems. These subsystems provide the capabilities needed to receive and use audio, video, and other data from a *broadcast server* on a client running the Microsoft® Windows® 98 operating system.

The following illustration shows an overview of the Broadcast Architecture subsystems.


Data Services subsystem
Television Client Services subsystem
TV Viewer subsystem
Broadcast Data Receiver subsystem
Broadcast Client Presentation subsystem

Each of these subsystems contains various components. Some of the components are supplied with Broadcast Architecture while others come from the Microsoft® Windows® 98 operating system or third party developers. In order to give a complete overview of the broadcast client, the following sections briefly describe all the components regardless of their source. The subsystems covered are:

- Broadcast Data Receiver Subsystem describes how data is presented to and received by the other subsystems.
- Data Services Subsystem describes the components that use the portion of the broadcast data stream that does not present audio or video.
- Broadcast Client Presentation Subsystem describes the Microsoft® DirectShow™ filter graph used in a broadcast client and the components that present audio and video.
- Television Client Services Subsystem describes components for configuring and controlling television channels.
- TV Viewer Subsystem describes the TV Viewer application and its components.

## Broadcast Data Receiver Subsystem

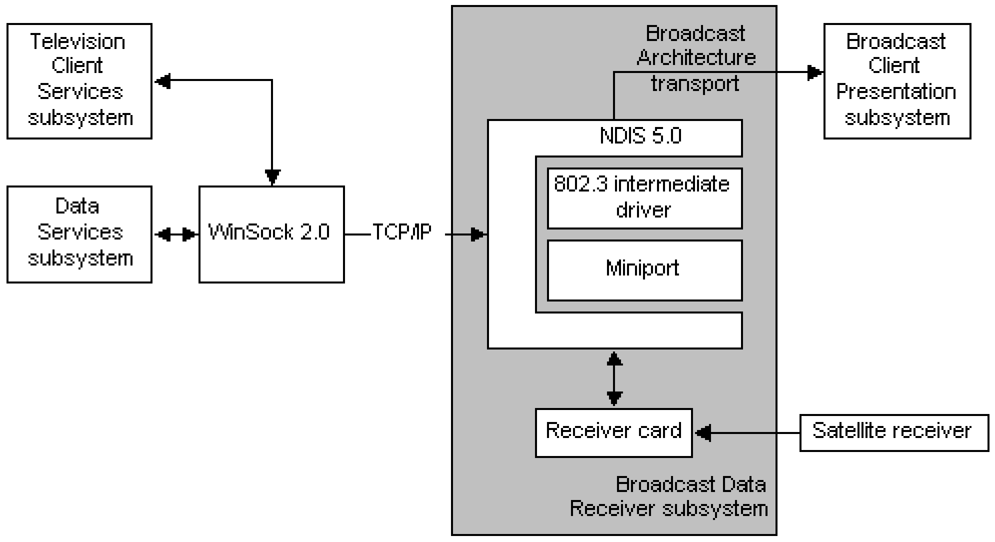
[This is preliminary documentation and subject to change.]

The broadcast client receives a number of different data streams over one or more broadcast networks. Drivers in the Microsoft® Windows® 98 operating system, called miniports, allow different hardware devices to have a common interface to the broadcast client. The miniport conforms to the Network Driver Interface Specification (NDIS) 5.0 including the Broadcast Architecture NDIS extensions. The use of NDIS 5.0 not only gives applications a common interface to the hardware, it also simplifies future porting of Broadcast Architecture to other NDIS compliant operating systems such as Microsoft® Windows NT®. For more information on Broadcast Architecture NDIS extensions, see the NDIS Extensions section of the Broadcast Architecture Device-Driver Kit (DDK), part of the device-driver documentation for Windows 98.

A Broadcast Architecture miniport splits the broadcast data into two streams. The first stream includes audio and video data sent by way of the Broadcast Architecture transport to the DirectShow filter graph. Any other data received, such as Hypertext Markup Language (HTML) pages, data files, or control information, is formatted as IP packets and sent to Windows Sockets (WinSock) version 2.0.

Applications that require a back channel to the server can connect to another network by using WinSock. Routing all data through WinSock simplifies the interface to the server by providing a single application programming interface (API) for data exchange.

The following illustration shows how the components of the broadcast data receiver fit together.


Television Client Services subsystem
Data Services subsystem
WinSock 2.0
TCP/IP
Broadcast Architecture transport
NDIS 5.0
802.3 intermediate driver
Miniport
Receiver card
Broadcast Data Receiver subsystem
Broadcast Client Presentation subsystem
Satellite receiver

The following topics briefly describe the components of the Broadcast Data Receiver subsystem.

## Windows Sockets

WinSock 2.0, a Windows API, is a Windows 98 system component. It provides a networking standard that gives applications an abstraction of the networking software below it. In the Broadcast Architecture, WinSock handles the computing data that may accompany the audio and video data streams.

## TCP/IP

Transmission Control Protocol/Internet Protocol (TCP/IP) is the industry-standard protocol used by WinSock 2.0 to send and receive data. WinSock packages and unpackages data into IP packets that it sends and receives to and from a TCP/IP network.

## Broadcast Architecture Transport

The Broadcast Architecture transport is a Broadcast Architecture component. It is a high-bandwidth transport with special support for features such as channel selection.The Broadcast Architecture transport allows various system components to move audio and video data through the system. For more information, see the Broadcast Architecture Transport section of the Broadcast Architecture Device-Driver Kit (DDK), part of the device-driver documentation for Windows 98.

## NDIS 5.0

NDIS 5.0 is a Windows 98 and Windows NT device driver standard that provides an operating system–independent standard for writing network device drivers. NDIS 5.0 allows independent hardware vendors (IHVs) to extend existing drivers to create hardware specific drivers. The extensions written by an IHV are called a *miniport*. In Broadcast Architecture, the NDIS-compliant miniports provide an interface between the network interface cards (NICs) and other broadcast client components.

## Broadcast Architecture Miniports

A Broadcast Architecture NIC miniport component, provided by independent hardware vendors, is the adapter-specific portion of an NDIS 5.0 driver. Each type of NIC requires a unique miniport. For more information, see the NIC Miniport section of the Broadcast Architecture Device-Driver Kit (DDK), part of the device-driver documentation for Windows 98.

## NDIS 802.3 Intermediate Driver

The NDIS 802.3 intermediate driver is a Broadcast Architecture component that translates *Multipacket Transport* (MPT) packets, from satellite networks, into *Internet Protocol* (IP) packets. This translation makes it possible for WinSock to handle the data as it does any other data carried by TCP/IP.

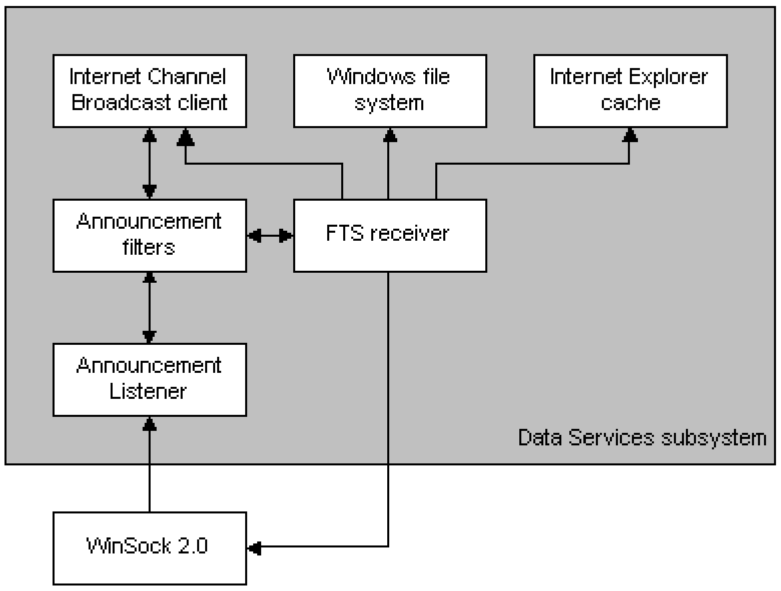
## Receiver Cards

A receiver card is a component provided by independent hardware vendors. This card receives incoming broadcast signals and converts them to data the computer can use. In Broadcast Architecture, it provides a point of entry for broadcast data into the system.

# Data Services Subsystem

[This is preliminary documentation and subject to change.]

The Data Services subsystem receives computing data streams from broadcast servers by using Windows Sockets (WinSock) 2.0 and routes that data to applications. Using Data Services over a broadcast network allows data to be sent to many users without the need for extra bandwidth that is required by other networks. Instead of sending separate packets to each client that requests data, broadcast servers send one packet that is received by all clients that are listening for the data. The Data Services subsystem listens for broadcasts and stores the data on the client computer. The following illustration shows how the Data Services components fit together.



## Announcement Listener

The Announcement Listener is a Broadcast Architecture component that monitors incoming announcements of upcoming transmissions. The Announcement Listener maintains a list of announcement filters that it uses to determine what to do with a given announcement. The announcement filter can handle the announcement itself or it can indicate that it is an announcement the Announcement Listener should handle. In this case, the Announcement Listener uses the information in the announcement to set up *File Transfer Service* (FTS) to receive data.

Announcements indicate when the data will be sent. If the data directly follows the announcement filter, the Announcement Listener starts the FTS receiver immediately. Otherwise, it can use the Task Scheduler in the Microsoft® Windows® 98 operating system to start the FTS receiver at the time indicated in the announcement.

## Announcement Filters

Announcement filters are supplied both as components in Broadcast Architecture and from independent software vendors. These filters are Automation servers used by Announcement Listener to determine which types of data broadcast to receive. A typical action for an announcement filter is to schedule the FTS receiver to receive a file.

## FTS Receiver

The FTS receiver application is a Broadcast Architecture component that receives data from a broadcast network. A typical use of the FTS receiver is to receive file packages for the Internet Channel Broadcast client. Other uses of the FTS receiver are to receive files and database updates. Depending on options passed to it, FTS can save the data to a disk, pass the data to another application, or if the data is an executable file load the data into memory and execute it.

## Internet Channel Broadcast Client

The Internet Channel Broadcast client is a Broadcast Architecture component that receives information from an Internet Channel Broadcast server. The information that the client receives can be Hypertext Markup Language (HTML) pages from the World Wide Web or data files. The Internet Channel Broadcast client includes an announcement filter that checks for announcements about upcoming data transmissions. A data file called Subscr.dat is a list of the data transmissions the client should receive.

After the broadcast client receives the data, it runs an unpackaging utility that copies the files into the Internet Explorer cache or to a directory given in the announcement.

## Internet Explorer

Internet Explorer is a Web browser included in Windows 98. Internet Explorer stores Web pages it has received in a cache on the local harddrive. The Internet Channel Broadcast client puts pages into this cache so the user can see them.

Broadcast Architecture includes Microsoft® ActiveX™ controls for showing video in Web pages displayed by Internet Explorer.

## Windows File System

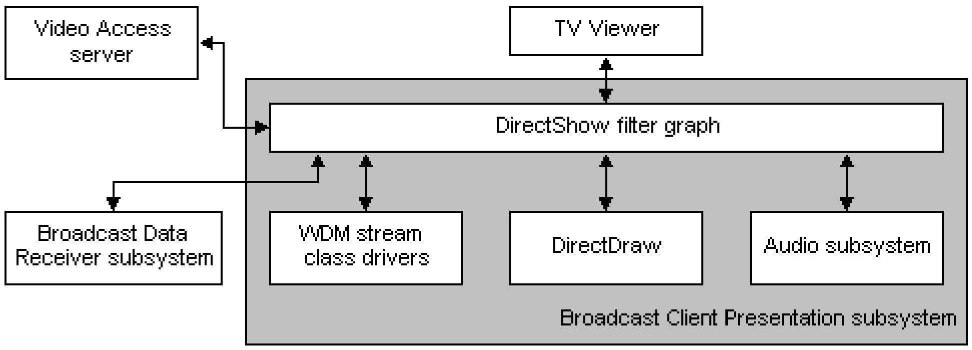
The Windows file system is simply the data storage system used by Windows 98.

# Broadcast Client Presentation Subsystem

[This is preliminary documentation and subject to change.]

Broadcast clients can work with data streams from a wide range of video and audio devices. To do this, Broadcast Architecture includes Microsoft® DirectShow™, a component of the Microsoft® Windows® 98 operating system that lets applications control high-bandwidth data streams.

Broadcast Architecture applications such as TV Viewer use a DirectShow filter graph to control the presentation of audio and video data. The following illustration shows how the components of the Broadcast Client Presentation subsystem fit together.



## DirectShow Filter Graph

The DirectShow filter graph is a Broadcast Architecture component that tells direct show what filters to use and how they are connected to each other. The filters for Broadcast Architecture represent Windows Driver Model (WDM) stream class drivers for audio and video devices. Each filter has *pins* that represent inputs, outputs, and controls for the driver. The filter graph shows how these pins are connected to each other and to applications that use the drivers.

## DirectDraw

The Microsoft® DirectDraw® application programming interface (API) is a Windows 98 dynamic-link library (DLL). Broadcast clients use a new feature of DirectDraw called Video Port Extensions (VPE). VPE provides an application programming interface (API) that accelerates graphics by providing direct access to bitmaps in off-screen display memory, as well as extremely fast access to hardware.

## Audio Subsystem

The audio subsystem is a component of Windows 98 that provides an API to applications to play audio.

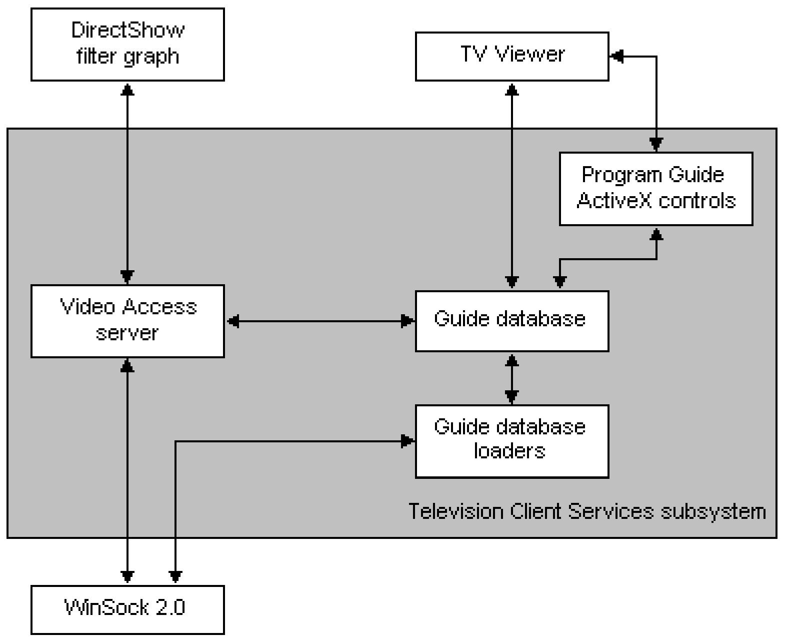
## WDM Stream Class Drivers

WDM stream class drivers are components of Windows 98 used to handle high-bandwidth data streams. Filters in the DirectShow filter graph let applications control these filters. An important feature of stream filters is that they can be connected directly to each other to reduce the overhead of handling the stream data. For example, an audio stream driver for a satellite receiver can be connected directly to a audio stream driver for a sound card. This direct connection enables the audio to play without requiring an application to copy data from one device to another.

# Television Client Services Subsystem

[This is preliminary documentation and subject to change.]

The following sections briefly describe the components of the Television Client Services subsystem. This subsystem includes a database called the Guide database. Other components in this subsystem use the Guide database to control TV Viewer and DirectShow filter graph. The following illustration shows how the Television Client Services components fit together.



## Video Access Server

The Video Access server is a Broadcast Architecture component. It runs on the client computer as a separate process that handles device contention among multiple instances of the TV Viewer *ActiveX control*.

## Program Guide ActiveX Controls

The *Program Guide* Microsoft® ActiveX™ controls are Broadcast Architecture components. These controls display the user interface to the Program Guide. For information on the Program Guide, see Overview of Program Guide Services.

## Guide Database

The Guide database is a Broadcast Architecture component. A Microsoft® Jet database, it stores current information about broadcast programs from various broadcast networks. For more information on the Guide database, see Overview of Program Guide Services.

## Guide Database Loaders

Guide database loaders are Broadcast Architecture components. Television Client Services also supports the use of third-party loaders. Generally, a vendor of Program Guide information provides a third-party loader to the broadcast client when a user subscribes to that vendor's service. These loaders process incoming Program Guide data and load it into the Guide database. The Broadcast Architecture Programmer's Reference includes a skeleton loader as an example for third-party developers. For more information on database loaders, see Updating Guide Data.

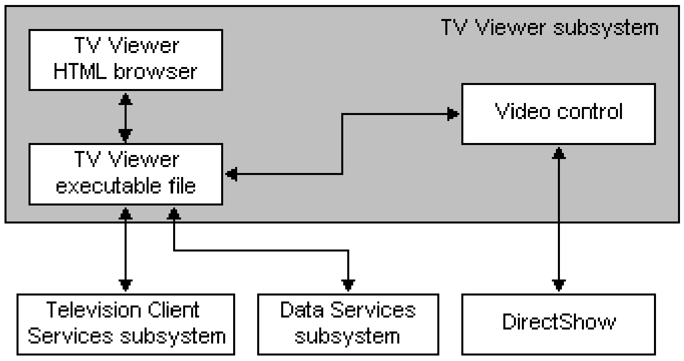
## TV Viewer Subsystem

[This is preliminary documentation and subject to change.]

TV Viewer is a Broadcast Architecture application. It displays enhanced television programs and their enhancements, if any.

The user first selects a channel using the Program Guide ActiveX controls in the Television Client Services subsystem. Then TV Viewer displays the program being broadcast on the selected channel. TV Viewer uses an *ActiveX control* to display this conventional television program. It also hosts an HTML browser to display a program's enhanced content.

The following illustration shows how the TV Viewer components fit together.



### TV Viewer Executable File

Tvx.exe is a Broadcast Architecture component. It is the application that users run to see television programs, both enhanced and unenhanced.

### TV Viewer HTML Browser

The TV Viewer HTML browser is a Broadcast Architecture component. It displays enhancements received as HTML pages from the broadcast data stream.

### Video Display Controls

The video display controls present audio and video on the screen. The Video control presents audio and video in a variety of environments, such as Web pages and stand-alone applications. The Enhancement Video control shows video and audio on a Web page displayed by TV Viewer. For more information about these controls and when to use them, see Which Video Control Should I Use?

# Client Hardware Requirements

[This is preliminary documentation and subject to change.]

The following sections explain the minimum and recommended hardware and operating system requirements for a *broadcast client*, including the Microsoft hardware manufacturing policy for such a client. These requirements extend the Microsoft PC 97 initiative to include requirements for a broadcast client, as follows:

- Minimum Configuration
- Recommended Configuration
- Windows 98 Requirements
- Hardware Manufacturer Policy
- Client Hardware Architecture

The following sections detail the hardware requirements for each of the components of a broadcast client:

- Network Receiver Card
- MPEG Decoding and Video Display
- Sound Card
- Monitor
- Input Devices
- Modem
- Related Hardware Information

## Minimum Configuration

[This is preliminary documentation and subject to change.]

The *minimum* hardware requirements for a broadcast client are:

- Intel Pentium 120 megahertz processor or compatible
- High-speed bus with slots adequate to accommodate high-speed broadcast network and video cards, such as PCI or Universal Serial Bus (USB)
- 16 megabytes (MB) RAM
- 1 gigabyte hard disk
- 27-inch super Video Graphics Array (SVGA) monitor capable of 640 x 480 resolution with a noninterlaced refresh rate of 60 hertz
- 3.5-inch 1.44 MB floppy disk drive
- Quad-speed CD-ROM drive
- Internal or external modem with a speed of 14,400 baud and compatible with Telephony

Application Programming Interface (TAPI)
- Standard computer keyboard
- Pointing device with two buttons

For more information on the minimum requirements for the Monitor, the Network Receiver Card, the Sound Card, and other components of the system, as well as Video Card Recommendations, see the sections detailing each component.

## Recommended Configuration

[This is preliminary documentation and subject to change.]

The recommended hardware configuration for a broadcast client is:

- Intel Pentium 150 megahertz processor.
- PCI bus with at least four slots available.
- Computer case of the consumer electronics type, with low-noise fan or noiseless cooling system. A user should not perceive broadcast client noise from a distance of 6 feet in a quiet living-room environment.
- Support for Simply Interactive PC (SIPC) initiatives such as OnNow, Drive Bay, 1394, and Universal Serial Bus (USB) to provide a more consumer-friendly appearance for the client.
- 16 megabytes (MB) RAM or more.
- Hard disk of 2 gigabytes (GB) or larger with a fast data transfer rate.
- 3.5-inch 1.44 MB floppy drive.
- 6x-speed CD-ROM drive or digital video disk (DVD).
- 31-inch monitor capable of 800 x 600 resolution with a noninterlaced refresh rate of 60 hertz.
- Internal fax modem with a speed of 28,800 baud or higher that is compatible with the AT command set. Modem functionality can be incorporated on expansion cards for the broadcast client, so the modem does not need to be a separate peripheral.
- Wireless keyboard that responds to radio or infrared frequencies, that is battery-operated, and that has a built-in pointing device.
- Wireless, television-style remote control.
- Combination remote control and wireless mouse with power and sleep button, TV buttons (channel up and down, volume up and down, and mute. This remote control/mouse should be battery operated.
- Sound-system expansion card with:
  - Digital audio, specifically Pulse Code Modulated (PCM) digital input
  - MIDI port
  - MIDI-controlled wave-table synthesizer
  - Multiple analog and digital audio inputs
  - Audio mixer and preamplifier that is software-controllable and low-noise
  - Multiple audio outputs

  Sound-card functionality can be incorporated on motherboard or expansion cards for the broadcast client, so the sound card does not need to be a separate peripheral.

- Built-in microphone or a front-mounted microphone jack suitable for teleconferencing, education, karaoke, and other applications requiring sound input.
- Audio compression using the AC-3 algorithm for DVD compatibility.
- Infrared remote control capable of controlling consumer electronic devices.

For more information on the recommended configuration for the Monitor, the Network Receiver Card, the Sound Card, and other components of the system, as well as Video Card Recommendations, see the sections detailing each component.

## Windows 98 Requirements

[This is preliminary documentation and subject to change.]

Because Broadcast Architecture uses the Microsoft® Windows® 98 operating system, hardware designed for it must conform to the Plug and Play architecture. For detailed information on designing hardware compliant with Plug and Play, see Plug and Play Specifications.

## Hardware Manufacturer Policy

[This is preliminary documentation and subject to change.]

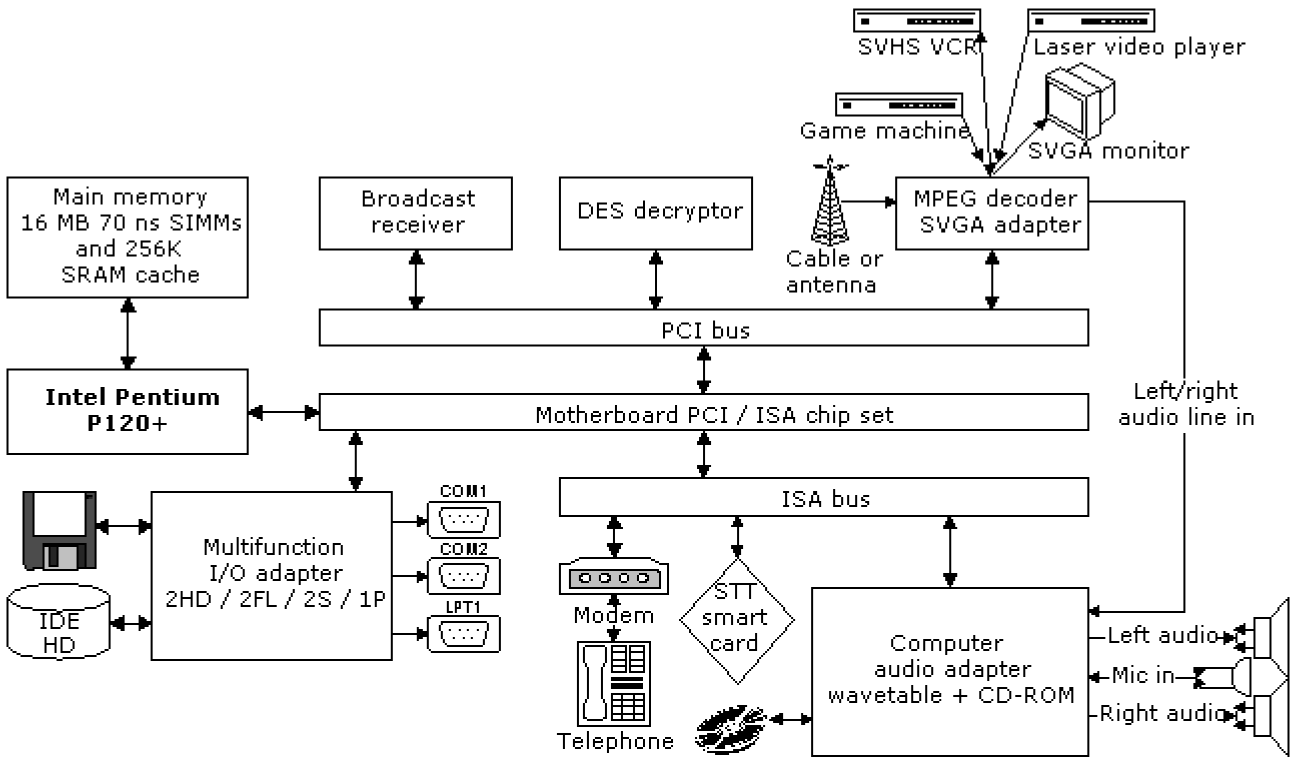
Broadcast Architecture components are not sold directly to the end user, with possible future exceptions being keyboards and pointing devices. Rather, OEMs and hardware vendors license the technology from a network provider, where applicable. OEMs also license *Microsoft Broadcast Data Network* (MSBDN) designs and services from Microsoft.

OEMs and hardware vendors are free to build, sell, install, and service their own products compatible with Broadcast Architecture. Because hardware designs and software components are available to numerous OEMs, competition is expected to drive the development of many feature enhancements, and to create distinct price points from which the end user can choose.

## Client Hardware Architecture

[This is preliminary documentation and subject to change.]

The following illustration shows how the data flows through the various components of the broadcast client hardware. In this illustration, SVHS is super VHS, SVGA is super Video Graphic Array, MB is megabyte, ns is nanosecond, K is kilobyte, SRAM is static RAM, DES is *Data Encryption Standard*, *MPEG* is the video standard, I/O is input/output, IDE is integrated device electronics, and STT is Secure Transaction Technology.



# Network Receiver Card

[This is preliminary documentation and subject to change.]

The following sections describe the functionality of a *broadcast receiver card* and the requirements it must meet to support Broadcast Architecture:

- Receiver Card Functionality
- Network Interface Connector or Antenna
- Signal Paths
- Receiver Card Requirements
- Other Networks

## Receiver Card Functionality

[This is preliminary documentation and subject to change.]

Because Broadcast Architecture works with many different types of broadcast digital networks, the exact details of a broadcast receiver card depend on the broadcast network and on network-specific access-control mechanisms. Broadcast Architecture software requires a specific software driver for each card. The card vendor or network provider supplies this driver. For more information on such drivers, see the NIC Miniport section of the Broadcast Architecture Device-Driver Kit (DDK), part of the device-driver documentation for the Microsoft® Windows® 98 operating system.

## Network Interface Connector or Antenna

[This is preliminary documentation and subject to change.]

For a broadcast satellite network, the antenna is typically an outdoor unit. For a cable network interface, the connector is likely a typical F connector used with standard television cabling. For a wireless cable system that supports the *Multichannel Multipoint Distributed System* (MMDS), a microwave antenna is required.

## Signal Paths

[This is preliminary documentation and subject to change.]

The broadcast receiver card must be able to receive both standard broadcast information from broadcast networks and data stream information, as defined by the MSBDN format for data transmission, which supports *Multipacket Transport* (MPT) and *Internet Protocol* (IP). For more information on this format, see the MSBDN Receiver section of the Broadcast Architecture Device-Driver Kit (DDK), part of the device-driver documentation for Windows 98.

Data streams are likely to be on channels, for example cable and Multichannel Multipoint Distributed System (MMDS) channels, separate from video and audio streams. Similarly, data streams are likely be on transponders, for example transponders for digital broadcast systems, separate from video and audio streams. This separation being usual, the broadcast receiver card requires two paths and signal lines. This section refers to these two paths as the "digital audio-video signal path" and the "digital data signal path." The following sections describe these signal paths separately to clarify their differences.

It is anticipated that hardware vendors will eventually combine both signal paths on the same PCI-bus card and also provide two separate tuners on one PCI-bus card. Such a combination means control, interface, bus mastering, power, and antenna connector resources can be shared. However, early versions of the broadcast receiver card need include only one tuner.

One-tuner versions of the card must receive both data and audio-video signals without user intervention. However, one-tuner cards need not receive both data and audio-video signals simultaneously, unless the signals are broadcast on the same channel or transponder. This reception functionality requires the card to include both network access control circuitry and MSBDN circuitry. The receiver card can also include a *smart card* as appropriate.

**Digital Audio-Video Signal Path**

The digital audio-video signal path contains network-specific technology for tuning, demodulating, decoding, error-correcting, demultiplexing, decrypting, and controlling access to digital audio and video signals.

The digital audio-video signal path must be capable of receiving at least four substreams simultaneously, such as video, audio, data, and Program Guide substreams. The design of the receiver card must allow transfer of these streams to computer memory with very low CPU utilization (less than 10 percent). Such a design ensures that the transfer does not interfere with the performance of applications running on the computer. This design implies that the card uses bus mastering or other direct memory access techniques.

Certain network designs mandate that particular access control functionality be resident in hardware. Such access control functionality controls some types of interaction with the consumer, for example preventing certain users from viewing video of specified ratings. In the broadcast client system, the software that verifies access for a particular network is divided into a number of parts. Security functions typically run on a microprocessor on the receiver card. The user interface and any modem interface functions are implemented in the computer. The different verifier portions communicate with each other through Broadcast Architecture driver interfaces; these interfaces manage verifier communication with the user and the network authorization center.

Digital audio-video data is tuned, demodulated, and error-corrected by a module referred to as the transport. The transport then routes the data to circuitry that selects portions of interest from the total stream. These portions may be identified by packet identifiers (PIDs). Information of interest can be passed through MSBDN circuitry for *Data Encryption Standard* (DES) decryption. After any required decryption processes, data is bus mastered into CPU memory.

Note that not all data is encrypted by the sender. Such data can bypass the circuitry for DES decryption.. It is also possible to take selected data directly from the error correction circuitry and bus master it into memory.

**Digital Data Signal Path**

The signal path that receives digital data is similar to the existing audio-video path, but it must meet the extra requirement that the digital data signal path must be able to demultiplex a packet stream into one or more substreams. In other words, it must be able to separate several combined data streams. During this demultiplexing process, the path must be able to filter out unwanted packets based on a field in the packet header.

For a more complete description of the data receiver and of required decryption functionality, see the MSBDN Receiver section of the Broadcast Architecture Device-Driver Kit (DDK), part of the device-driver documentation for Windows 98.

## Receiver Card Requirements

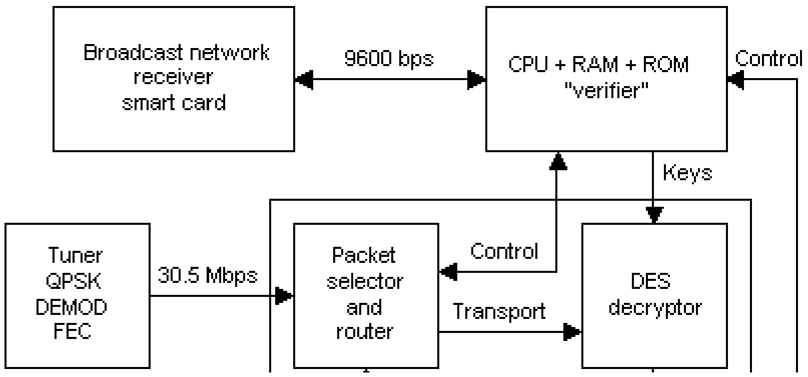
[This is preliminary documentation and subject to change.]

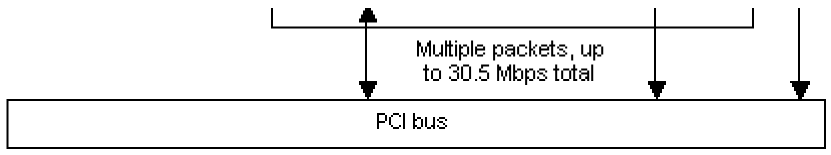
The receiver card's interface with the computer should:

- Provide a mechanism for moving data into the computer and for specifying the destination for that data in computer memory, preferably through bus mastering.
- Control the tuner and retrieve tuner information.
- Control decoding of video, audio, and other data.
- Control the different Viterbi or other decoding rates used by the digital broadcast network.
- Control what packet identifiers (PIDs) are received and what data is routed to the computer.
- Retrieve status information about errors and control error correction.
- Satisfy the requirements for receiving MSBDN packets, which can require additional hardware beyond that required for audio and video. For more information on receiving MSBDN packets, see the MSBDN Receiver section of the Broadcast Architecture Device-Driver Kit (DDK), part of the device-driver documentation for Windows 98.
- Support of at least five PIDs simultaneously. Support of eight is recommended.
- Perform one of the two following tasks:
  - Present an advancing 27 megahertz register, a register containing the last-received system clock reference (SCR) or other reference time stamp, and a register containing the value of the 27 megahertz register when the last SCR was received.
  - Generate an interrupt immediately upon receipt of each SCR and have that SCR read through the PCI bus.

- Perform PCI bus mastering with support for scatter/gather memory access and unaligned, odd-byte memory transfers. This requirement includes support for time-critical MPEG packets of 127 bytes and less.

The following illustration shows the receiver card's internal and external data flow. In this illustration, bps is bits per second, QPSK is Quadrature Phase Shift Keying (a method of encoding digital data in an analog signal), DEMOD is demodulator, DES is Data Encryption Standard, and Mbps is megabits per second.


Broadcast network receiver smart card
9600 bps
CPU + RAM + ROM "verifier"
Control
Keys
Tuner QPSK DEMOD FEC
30.5 Mbps
Packet selector and router
Control
Transport
DES decryptor



## Other Networks

[This is preliminary documentation and subject to change.]

In the future, cards very similar to the broadcast receiver card will probably be built for the cable, digital video disk, *Asymmetrical Digital Subscriber Line* (ADSL), and Multichannel Multipoint Distributed System (MMDS) environments. Because these network cards will receive broadcast data rather than individually targeted data, and because they have no back-channel requirement, they will require minimal support from a network head end.

# MPEG Decoding and Video Display

[This is preliminary documentation and subject to change.]

The following sections describe the functionality of an *MPEG* decoder circuit and a video crossbar circuit, and the requirements they must meet to support Broadcast Architecture:

- MPEG Decoder Functionality
- MPEG Decoder Requirements
- MPEG Decoder Recommended Configuration
- MPEG Decoder Interface
- Analog Audio and Video Interconnections
- Integration Possibilities

The components described in these sections are part of a video system that includes standard video cards and sound cards. For functionality recommendations for the standard video card for Broadcast Architecture, see Video Card Recommendations. For functionality recommendations for the standard sound card for Broadcast Architecture, see Sound Card Functionality.

## MPEG Decoder Functionality

[This is preliminary documentation and subject to change.]

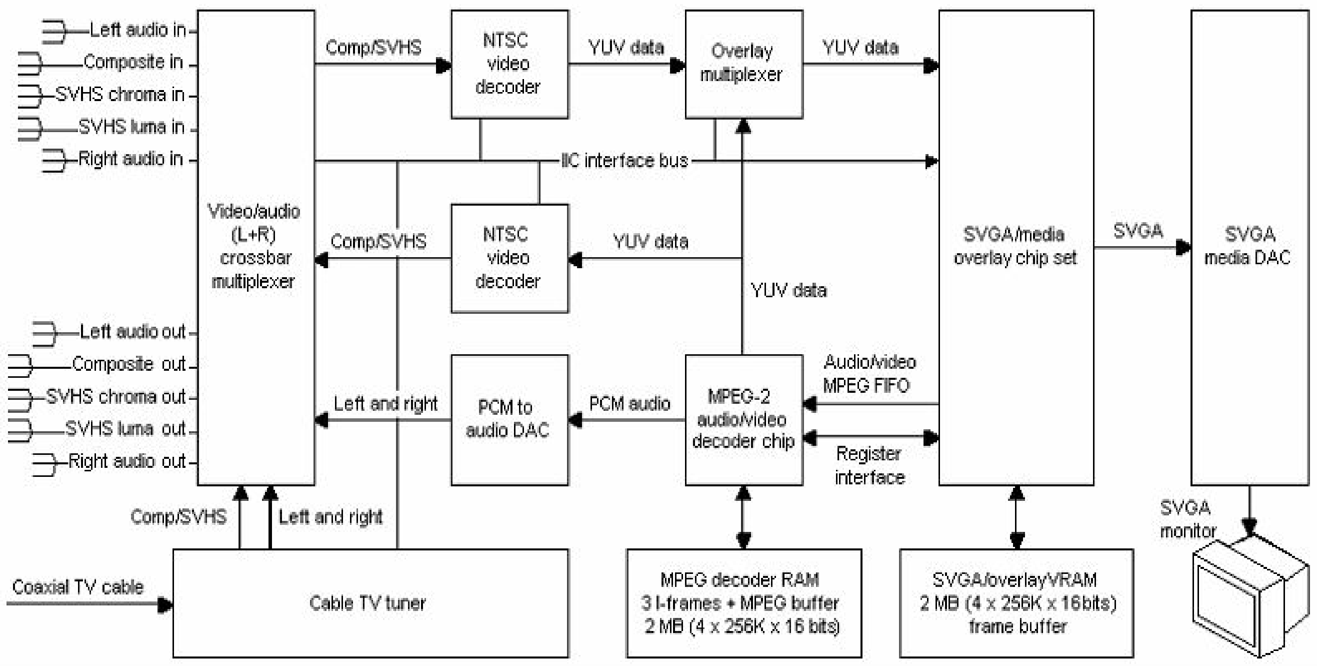
The MPEG decoder is a video and audio decoder capable of handling MPEG-2 main profile and main level data rates (15 megabits per second). The decoder combines MPEG video with standard computer video for display on a super Video Graphics Array (SVGA) monitor. The MPEG decoder has 2 megabytes (MB) RAM for SVGA, including the *frame* buffer, and another 2 MB RAM for the MPEG decoder itself.

The computer supporting the decoder should also offer a television tuner and base-band, *S-video* inputs. It should also offer outputs for routing *NTSC* and *PAL* signals to and from VCRs, laser disk players, and other video sources. In addition, the computer must provide encoder and decoder circuitry. This circuitry converts NTSC or PAL input signals into digital data in the video frame buffer and converts decoded MPEG video to NTSC or PAL output signals.

The decoder computer must provide separate audio and video outputs that carry audio and video from video sources, as opposed to computer-generated music and sound effects. These separate outputs are provided so that video and its associated audio can be recorded without recording computer-generated music and sound effects. Associated circuitry provides recording control features that can be activated and deactivated under software control. This recording control includes Macrovision encoding of NTSC video generated directly from the MPEG decoder.

The video card portion of the MPEG decoder must meet the criteria of the Plug and Play Framework architecture that is part of the Microsoft® Windows® 95 operating system. Plug and Play provides specific methods for extending the VGA register set and requirements for the appearance of expanded video-card resources on the system bus. For more information on Plug and Play, see Plug and Play Specifications.

The following diagram shows a video system that includes an MPEG decoder. In this illustration, Comp is composite; SVHS is super VHS; IIC is Inter-Integrated Circuit; SVGA is super Video Graphics Array; DAC is digital to analog converter; PCM is Pulse Code Modulated digital input; FIFO is first in, first out; MB is megabytes; K is kilobytes; and VRAM is video RAM.



## MPEG Decoder Requirements

[This is preliminary documentation and subject to change.]

The video card for the MPEG decoder must have the following features:

- Super Video Graphics Array (SVGA) card capable of at least 800 x 600 x 8 bits per pixel, with a 60 hertz refresh rate.
- Support for high-speed, two-dimensional graphics acceleration, preferably with support for three-dimensional bitmaps and MPEG textured polygons.
- MPEG-2 decoder chip set, capable of a speed of 15 megabits per second and a screen resolution of 720 x 480 x YUV 4:2:2.
- Data transfer using bus mastering from memory to the MPEG decoder, at up to 2 megabytes per second in speed. Such transfer must include support for unaligned, odd byte transfer and scatter/gather memory access.
- Capture and display of each 720 x 240 video *field* stretched vertically to 720 x 80 without decimation.
- Broadcast cable television tuner that supports *NTSC* or *PAL* signals.
- Composite super VHS (SVHS) – type chip that decodes NTSC or PAL to *YUV*, with an interface to the SVGA chip set.
- Composite SVHS-type chip that encodes data to NTSC or PAL, with a direct interface from the MPEG decoder.
- Audio and video crossbar switching chips for interconnecting composite and SVHS inputs and outputs.
- SVGA format that supports capture of YUV frame buffers for prioritized overlay and scaling with horizontal and vertical interpolation.
- Merging of YUV video frames and SVGA video frames in the digital to analog converter (DAC) using chroma and *color keying*.
- Controllable Macrovision video encoding of all composite video signals that are output.
- Decoding of *vertical blanking interval* (VBI) data for NTSC format, including decoding of such information as closed captioning, from all composite inputs.
- Encoding of NTSC VBI data (such as closed captioning or time codes) on all composite outputs.
- Full driver support for the Microsoft® Windows® 98 operating system, including display driver and MPEG minidriver support.
- Support for the output of audio from video sources to the line input of standard audio cards.

## MPEG Decoder Recommended Configuration

[This is preliminary documentation and subject to change.]

The recommended video card for the MPEG decoder has the features listed in MPEG Decoder Requirements, plus:

- Super Video Graphics Array (VGA) card capable of at least 800 x 600 x 24 bits per pixel with a 60 hertz refresh rate
- Hardware acceleration compatible with the Microsoft® Direct3D® application programming interface (API)
- Broadcast cable television tuner that supports *NTSC*, *PAL*, and *SECAM* in foreign and domestic stereo versions
- Multiple composite and *S-video* jacks
- Optional decoding of arbitrary *vertical blanking interval* (VBI) data for NTSC, such as Intercast, from all composite inputs

## MPEG Decoder Interface

[This is preliminary documentation and subject to change.]

The interface between the video card for the MPEG decoder and the computer must have the following features:

- Conformation to the Plug and Play specification as a PCI multiple-function device, providing separate spaces for super Video Graphics Array (SVGA), MPEG, and tuner functions
- PCI bus mastering of audio and video MPEG data
- Support for unaligned, odd byte transfers and scatter/gather memory access
- Data buffering control, and provision of status information about data buffering
- MPEG video decoding control, and provision of status information about MPEG video decoding
- MPEG audio decoding control, and provision of status information about MPEG audio decoding
- Cable television tuner control, and provision of status information about the cable television tuner
- Control of and provision of status information about base-band video encoding and decoding
- Control of and provision of status information about audio and video signal paths
- Closed captioning input and output to audio and video signal paths

## Analog Audio and Video Interconnections

[This is preliminary documentation and subject to change.]

If a video card supports analog connections, it should allocate connections of input sources to output sinks. Actual device connections vary somewhat based on consumer equipment. However, some connections of inputs to outputs are fixed, in that they interconnect specific devices, such as computer encoders to decoders. There are many possible input sources and output sinks for analog connections, such as a VCR for playing, a VCR for recording, a laser disk player, a camcorder, a game console box, a cable box, a cassette tape deck, and an FM audio tuner.

The following table shows suggested allocations of input lines to output lines for video and matches video input sources and output sinks with typical uses.

| **Video input (8 sources)** | **Typical input** | **Video output (6 sinks)** | **Typical output** |
|---|---|---|---|
| Composite 1<br>*S-video* Y 1<br>S-video C 1 | Computer MPEG *NTSC* TV encoder | Composite 1<br>S-video Y 1<br>S-video C 1 | Computer NTSC decoder |
| Composite 2<br>S-video Y 2<br>S-video C 2 | Super VHS (SVHS) VCR | Composite 2<br>S-video Y 2<br>S-video C 2 | SVHS VCR |
| Composite 3 | Computer TV tuner | | |
| Composite 4 | Camcorder | | |

The following table shows suggested allocations of input lines to output lines for audio and matches audio input sources and output sinks with typical uses.

| **Audio input (5 stereo sources)** | **Typical input** | **Audio output (4 stereo sinks)** | **Typical output** |
|---|---|---|---|
| Left 1 and right 1 | Computer MPEG or AC-3 pulse code modulation (PCM) digital to analog | Left 1 and right 1 | Computer sound board CD-ROM audio in |
| Left 2 and right 2 | SVHS VCR | Left 2 and right 2 | SVHS VCR |
| Left 3 and right 3 | Computer TV tuner | Left 3 and right 3 | Surround sound audio processor |
| Left 4 and right 4 | Camcorder | Left 4 and right 4 | Headphones |
| Left 5 and right 5 | Computer CD-ROM audio wire | | |

## Integration Possibilities

[This is preliminary documentation and subject to change.]

Some of the possible audio and video card configurations:

- Super Video Graphics Array (SVGA) card capable of PCI bus mastering, with an integrated MPEG-2 audio and video decoder, audio-video switching, and an ISA audio adapter compatible with SoundBlaster
- SVGA card capable of PCI bus mastering, with an integrated MPEG-2 audio and video decoder, a SoundBlaster chip set, and audio-video switching

**Note** Either of these configurations works, but they have significant tradeoff issues in terms of the number of cards and the use of standardized cards. In general, it is better to have fewer cards and to use more standard cards.

# Sound Card

[This is preliminary documentation and subject to change.]

The following sections describe the functionality of a sound card and the requirements it must meet to support Broadcast Architecture:

## Sound Card Functionality

[This is preliminary documentation and subject to change.]

Sound components can be laid out on a separate sound card or integrated elsewhere. If sound is laid out on a separate card, the television and *MPEG* sound from the video card must be connected to the sound card's line-in port, and this sound must be controlled with the standard mixer interface in Microsoft® Windows® operating systems. The video card selects the television-related sound that is heard by users, and the mixer controls the television volume.

For game support on a broadcast client, sound components compatible with SoundBlaster that provide wave table synthesis are recommended. The sound components must be capable of a signal-to-noise ratio of 90 decibels, to maintain the audio quality of the digital broadcast network. It must be possible to use the sound-card circuitry independent of the digital broadcast audio output — for example, when the user tape-records a program while playing a video game.

Although sound and video display components can be laid out on separate cards, achieving the highest possible sound quality usually requires a digital connection between the video display card and the sound card.

For compatibility with digital video disk (DVD) audio, an audio decoder with compression to the AC-3 algorithm is recommended.

## Sound Card Requirements

[This is preliminary documentation and subject to change.]

The sound card must have the following features:

- Stereo
- SoundBlaster compatibility
- Signal-to-noise ratio of 90 decibels
- Ability to satisfy the sound requirements of broadcast networks, including digital broadcast networks
- Ability to mix audio from the computer, the current video source, and reference CD-ROMs

## Sound Card Recommended Features

[This is preliminary documentation and subject to change.]

The recommended sound card for the MPEG decoder has the features listed in Sound Card Requirements, plus:

- Wave-table synthesis
- Separate mixer volume controls for tuner and MPEG audio output
- Support for audio compressed to the AC-3 algorithm

# Monitor

[This is preliminary documentation and subject to change.]

The following sections describe the attributes a monitor must possess and the requirements it must meet to support Broadcast Architecture:

- Monitor Functionality
- Monitor Requirements
- Recommended Features for the Monitor

For information about video cards for Broadcast Architecture, see Video Card Recommendations

## Monitor Functionality

[This is preliminary documentation and subject to change.]

Monitors for a broadcast client should have the following key attributes:

- Support for the minimum resolution, currently 640 x 480. However, 800 x 600 resolution is recommended. Typical satellite digital broadcasters transmit main level and main profile *MPEG-2* — the middle level of the five possible levels of MPEG-2 encoding for video data. That transmission level translates into 720 x 480 x 30 frames per second for *NTSC*. For *PAL*, this level translates into 720 x 576 x 25 frames per second. As a result of this transmission type, display decimation occurs at 640 x 480 resolution, which is why 800 x 600 resolution is strongly recommended.
- Support for color gamma suitable for display of television and computer data. Diverting an MPEG stream to a computer monitor can cause color problems. These problems occur because the MPEG stream is encoded into an abstract color space, then decoded and sent to a computer monitor with a different, possibly greater color gamma than an NTSC display device.
- Picture tube that is ideal for both computers and televisions. Such a picture tube has high luminance, analog super Video Graphics Array (SVGA) inputs, medium phosphor persistence, and a progressive scan of 60 hertz.
- Good corner convergence.
- Large screen size. For an optimal viewing experience, it is recommended that OEMs build large-screen monitors for Broadcast Architecture, 31 or 33 inches in size measured on the diagonal. However, the broadcast functionality works on any size computer monitor, leaving the ultimate configuration up to the OEMs.
- Refresh rate of 60 hertz, or an integral multiple of 60 hertz, for any mode in which video is displayed. Most source video, such as NTSC or MPEG-2 video and film, is created or adjusted through temporal rescaling at a 3:2 ratio. This adjustment is performed expressly for 60 hertz television monitors. Further rescaling to other refresh rates, such as 72 hertz, introduces unacceptable motion artifacts, such as nonlinear screen motion. By using a refresh rate of 60 hertz or a multiple thereof, the monitor can be balanced with the monitor phosphor.

  For a flicker-free viewing experience, a medium-persistence phosphor should be used instead of a short-persistence phosphor. If the refresh rate is fixed at 120 hertz, the standard computer-monitor phosphors can be used.

## Monitor Requirements

[This is preliminary documentation and subject to change.]

The monitor must have the following features:

- 27-inch analog monitor that supports super Video Graphics Array (SVGA).
- Minimum of 640 x 480 resolution, although 800 x 600 resolution is strongly recommended, with a refresh rate of 60 hertz.
- Digitally controlled geometry: skew, rotate, pincushion, size, position.
- Support for the Standard DDC2B VGA Monitor Identification Protocol.
- Television-level brightness and phosphor luminance.
- Maximum 0.75-millimeter dot pitch for true 800 x 600 resolution at a monitor size of 27 inches. Proportionally higher dot pitch can be provided in larger monitors.
- Medium-persistence phosphors, optimized for 60 hertz with no noticeable flicker.
- Compliance with Energy Star power savings standard.

## Recommended Features for the Monitor

[This is preliminary documentation and subject to change.]

The recommended monitor has the features listed in Monitor Requirements, plus:

- Front panel digital controls for contrast, brightness, and degauss.
- Black packaging in styled plastic, suitable for consumer electronics.
- 31-inch or larger analog monitor that supports super Video Graphics Array (SVGA). Such 31-inch monitors are often labeled at 33 inches.
- Brightness and contrast controls revealed through a driver with a remote-control user interface, rather than through dials on the monitor case.
- Host control of the monitor by using video cards and drivers compatible with Display Data Channel (DDC) requirements.
- Adjustable color level appropriate to computer and video applications.
- Ability to select between power at 120 volts alternating current (VAC) and 60 hertz and power at 240 VAC and 50 hertz.

# Input Devices

[This is preliminary documentation and subject to change.]

For Broadcast Architecture, user input devices such as keyboards and mice change to accommodate their use in a living room setting. For example, keyboards can be used from the lap or a coffee table. Pointing devices can be hand-held, because a flat surface might not be available. Users can expect to use a remote control for channel surfing and other control of entertainment systems.

For further information on input devices suitable for Broadcast Architecture, see the following sections:

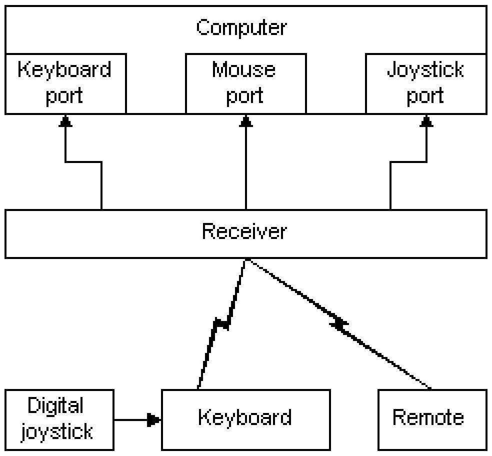
- Connection to the Computer
- Pointing Device/Remote Control
- Keyboard
- New Windows Keys

## Connection to the Computer

[This is preliminary documentation and subject to change.]

In Broadcast Architecture, one receiver for all remote controls connects to the computer through the keyboard, mouse, and possibly joystick ports. Alternatively, the receiver can use a driver supplied by its hardware vendor that emulates these ports. This design allows each remote device to send data to each computer port. For example, a remote control might act as a mouse but might also send special keyboard scan codes. (For more information on these codes, see New Windows Keys.)

The following illustration shows the relationship between the remote controls, the receiver, and the computer.



## Pointing Device/Remote Control

[This is preliminary documentation and subject to change.]

This section describes the functionality of a remote control and the requirements it must meet to support Broadcast Architecture.

**Functionality for the Pointing Device/Remote Control**

The integration of the pointing device and remote control are necessary so broadcast client users can navigate within computer applications, television programs, and television applications.

Consumers expect the pointing device/remote control to be very durable, with a long battery life. The type of pointing device used can be a directional keypad, a touch pad, a trackball, or some other device with similar functionality. The device must be hand-held. It can resemble a standard television remote control, with the addition of special keys to handle tasks for television and computer applications. For more information on these keys, see New Windows Keys.

**Requirements for the Pointing Device/Remote Control**

The pointing device/remote control must have the following features:

- Mouse compatibility.
- Long battery life.
- Numeric keys. Touch tone labeling of numeric keys, as used on telephones, is suggested.
- Television-related keys.
- Navigation keys.
- Compatibility with Simply Interactive PC (SIPC) remote-control guidelines, as described in *SIPC Remote Control Design Guide.*

## Keyboard

[This is preliminary documentation and subject to change.]

This section describes the functionality of a keyboard and the requirements it must meet to support Broadcast Architecture.

**Keyboard Functionality**

The broadcast client keyboard has the same functionality as a standard computer keyboard, with some additional support for special keys. For more information on these keys, see New Windows Keys.

Consumers expect the keyboard to be very durable, with a long battery life. The keyboard must be capable of use from the lap or a coffee table. It can possibly be integrated with a pointing device. If the keyboard is so integrated, radio frequency and infrared create one-way signals to the computer. Therefore, status lights cannot be controlled on the keyboard.

**Keyboard Requirements**

The keyboard must:

- Be wireless
- Have a long battery life
- Support Windows keys
- Support television-related keys
- Optionally, have an integrated pointing device

## New Windows Keys

[This is preliminary documentation and subject to change.]

In Broadcast Architecture, keyboards and possibly other input devices have the following additional keys defined by Microsoft and implemented on the Microsoft® Natural® keyboard:

- Windows (left)
- Windows (right)
- Application

# Modem

[This is preliminary documentation and subject to change.]

The following sections describe the functionality of a modem and the requirements it must meet to support Broadcast Architecture:

- Modem Functionality
- Modem Requirements
- Modem Recommendations

## Modem Functionality

[This is preliminary documentation and subject to change.]

The Broadcast Architecture modem is a standard computer-compatible modem. Broadcast Architecture uses a modem for the following purposes:

- To communicate billing information to service providers
- To supply an interactive back channel
- To provide dial-up connectivity for user applications, such as online services

## Modem Requirements

[This is preliminary documentation and subject to change.]

At a minimum, the modem must support the following:

- The Microsoft® Windows® 98 operating system, including Telephony Application Programming Interface (TAPI) and communications under the Microsoft® Win32® API. Most modems are compatible with Windows 98.

  A modem is a device used for data or fax transmission. In Windows 98, such a modem uses TAPI functions for call control and a combination of Win32 communications and Windows Sockets (WinSock) for data transactions.

  If Windows 98 does not support a modem, or the modem is not directly compatible with a supported model, the modem manufacturer must supply a driver for the modem. This driver must supply the Service Provider Interface (SPI) functions called by TAPI. However, from a practical standpoint, such drivers are rarely required. Most commercially available modems comply with Windows 98, because almost all comply closely with the international standards to use the UNIMODEM service provider included in Windows 98.

- Dial-up to the billing center for the broadcast network. The modem must be able to communicate with the network billing center.
- Dial-up connection by the user to online services and Internet service providers, such as the Microsoft® MSN™ online service. Communications speed of 9600 baud or higher is strongly recommended.

The modem must also provide the following capabilities as minimum requirements.

### Modem Operation Modes

- V.22 operation at 1200 baud
- V.22 bis operation at 2400 baud
- V.32 operation at 9600 baud

### Modem Protocol

•V.42 or V.42 bis

### Modem Command Set

•TIA-602 (Hayes compatible)

**Connectors**

The connection to the modem from the telephone line must be a standard RJ-11 jack.

## Modem Recommendations

[This is preliminary documentation and subject to change.]

These optional features, although not required for modem operation, enhance the modem's value and customer satisfaction:

- VoiceView technical support. This feature permits a voice call to be suspended while modem data exchanges take place. Generally, the primary users of this feature are telephone support personnel who want to interrogate or alter a customer's computer during a support call. Microsoft® Technical Support provides hardware and software tools to enable this feature for support of products for the Microsoft® Windows® 95 and Windows 98 operating systems.
- Capability to send and receive faxes. If the modem has this feature, it should possess at least class 1 fax capability, and it should be able to originate and receive group 3 fax calls.
- Packaging within the computer. While an external modem fulfills the minimum requirements for Broadcast Architecture, use of an internally mounted modem with an ISA bus simplifies user installation.

# Related Hardware Information

[This is preliminary documentation and subject to change.]

This following sections describe additional recommendations for and information about support for Broadcast Architecture:

- Video Card Recommendations
- Displaying Interleaved Video on Progressive Monitors
- Plug and Play Specifications

## Video Card Recommendations

[This is preliminary documentation and subject to change.]

The following features are recommended for video cards supporting Broadcast Architecture on computers running the Microsoft® Windows® 98 operating system:

- The standard Video Graphics Array (VGA) page frame and input/output (I/O) address resources should be static (that is, they cannot be relocated). The VGA basic input/output system (BIOS), if it exists separately, has the base address fixed at C000h.
- A linear frame buffer should be used. It must be possible to relocate this buffer above the 16 megabyte boundary by using software, where applicable.
- The card BIOS should meet the Display Data Channel 1(DDC1) host requirements documented in the Video Electronics Standards Association (VESA) DDC standard.
- The video card ROM or virtual display driver should support the VESA BIOS extensions for power management (that is, the VESA BIOS Extensions/Power Management Standard).
- Color ordering should be blue-green-red, with red as the high byte in displays supporting 16 bits per pixel and 24 bits per pixel. This ordering takes advantage of Windows 98 graphics capabilities.
- The VGA BIOS, if it exists separately, should be configurable to two addresses at a minimum.
- Any 24-bit or higher-bit displays should support downloadable entries in random access memory digital-to-analog converter (RAMDAC) format. Such support allows gamma correction in hardware.
- The card should connect to a high-speed expansion bus, such as a PCI bus.
- The card should be capable of supporting a monitor resolution of at least 800 x 600 by 8 bits per pixel.
- The VGA video plane should support *color keying*.
- If interrupt request 2 (IRQ2) is supported for VGA compatibility, it should be inactive when the computer is turned on. The computer should not claim IRQ2 as a static resource.
- If extended display resources are used, the card should at a minimum be able to map the I/O addresses to seven locations and to disable them. However, this functionality is not necessary if the extended resource addresses are aliases of the standard VGA addresses.
- A software-tunable crystal should be used, so that the 27-megahertz clock driving the *MPEG* chip can match the 27-megahertz clock from the broadcaster. Software drivers adjust the MPEG chip's frequency to match the rate captured by the broadcast receiver card. The range and number of steps for tunability are based on crystal accuracy and the requirements for *NTSC* color signals.
- Both super Video Graphics Array (SVGA) and MPEG video clocks should be based on the same tunable crystal. This functionality prevents skipped or doubled frames due to slight differences in frequency between two different crystals.
- For improved picture quality, the card should support horizontal and vertical interpolation as specified for Comité Consultatif International des Radiocommunications (CCIR) 601 video, 720 x 480 x YUV 4:2:2 screen resolution at 60 hertz.

For more detailed information about these features, refer to *Hardware Design Guide for Microsoft® Windows® 95*, available from Microsoft Press®.

Another important point to note is that both the MPEG decoder and the SVGA should be running at the same 60 frames per second to avoid beat frequency artifacts. One solution is to have the MPEG decoder and the SVGA share a single clock crystal tunable to a voltage-controlled crystal oscillator

(VCXO). This crystal is run through a phased lock loop to generate both the 27 megahertz decoder clock and the SVGA pixel clocks. This clock tuning is necessary to synchronize the clocks to the MPEG encoder's time base of 60 hertz.

## Displaying Interleaved Video on Progressive Monitors

[This is preliminary documentation and subject to change.]

The recommended way of deinterlacing and displaying *NTSC* video, or NTSC video encoded to MPEG, is to show each 720 x 240 x YUV field at 60 hertz on the SVGA screen. This display uses vertical and horizontal interpolation to get at least a full-frame image, that is a 720 x 480 x YUV image.

In this display, the alternating odd and even *fields* should be shown such that the odd fields are offset by one scan line from the even fields after scaling. Every other field is offset to get rid of image jitter when switching from one field to another. Cropping the extra top and bottom line of the offset fields is also recommended. This sequential display of offset odd and even fields on a 60-hertz (Hz), progressive scan SVGA monitor, running at 800 x 600 or greater resolution, simulates the way video fields appear on a traditional television screen that employs *interlacing*.

The following table illustrates this technique. Odd-numbered fields have the format of the first field; even-numbered fields have the format of the second field.

| **First field output (720 x 480 at 60 Hz)** | **Second field output (720 x 480 at 60 Hz)** | **Third field output (same as first field)** | **And so on...** |
|---|---|---|---|
| 0 (This line should be cropped) | | 0 (This line should be cropped) | |
| (0+2)/2 | 1 | (0+2)/2 | |
| 2 | (1+3)/2 | 2 | |
| (2+4)/2 | 3 | (2+4)/2 | |
| 4 | (3+5)/2 | 4 | |
| … | | | |
| (n-2 + n)/2 | n-3 | (n-2 + n)/2 | |
| n | (n-3 + n-1)/2 | n | |
| | n-1 (This line should be cropped) | | |

Each field is captured in the display buffer as a 720 x 240 x YUV, 2-byte video plane. The digital to analog converter (DAC) should perform interpolation. To do so, in the DAC a line store buffer that has first in, first out (FIFO) format is loaded. This buffer interpolates the lines of a field as the field is being output. Each video plane requires 345,600 bytes; two planes require 675 kilobytes of video memory.

Double buffering is required to prevent the user from seeing incomplete images, which occurs with a single buffer being updated as it is being output. The double buffers are swapped at the start of the vertical blanking interval (VBI) if and only if a new buffer has been filled with YUV data.

The fields are offset by one scan line, and the "extra" line on each field should be cropped. This cropping is performed because the extra line is only shown at 30 hertz, whereas the rest of the lines are refreshed at 60 hertz. The monitor should be run at 800 x 600 resolution at 60 hertz. This functionality means the fields are often stretched to fill the full display area but are sometimes shrunk, such as when video is displayed in a window.

Note that if a single horizontal line appears on only one field of the display surface, it flickers. All televisions have this problem with NTSC fields. The interlaced approach, though it produces this flicker, avoids various unpleasant artifacts such as feathering, tearing, and odd-field discarding.

## Plug and Play Specifications

[This is preliminary documentation and subject to change.]

Plug and Play technology makes it possible for computer hardware and attached devices work together automatically. With Plug and Play, a user can simply attach a new device and begin working without restarting the computer, even while the computer is running. Plug and Play technology is implemented in hardware, in operating systems, and in support software such as drivers and basic input/output systems (BIOS).

With Plug and Play technology, users can easily add new capabilities to their computers, such as sound or fax, without concerning themselves with technical details or problems. For users of mobile computers who frequently change configurations with docking stations and have intermittent network connections and so on, Plug and Play technology easily manages their changing hardware configurations. For all users, Plug and Play reduces the time wasted on technical problems and increases productivity and satisfaction with computers.

A variety of Plug and Play technologies currently exist, including BIOS, ISA, small computer system interface (SCSI), integrated drive electronics (IDE) CD-ROM, LPT, Component Object Model (COM), Personal Computer Memory Card International Association (PCMCIA), and Plug and Play drivers. Specifications are available for many of these technologies. However, in a nutshell, each hardware device supporting Plug and Play must:

- Be uniquely identifiable.
- State the services it provides and the resources it requires.
- Identify the driver that supports it.
- Allow itself to be configured by software.

For Broadcast Architecture, there are several hardware units that can reside on a video card, receiver card, or motherboard. Cards can also combine one or more functions usually provided by separate units. For example, a video card may expose separate functions for super Video Graphics Array (SVGA), *MPEG-2*, and video tuning.

Cards that include one or more functions should identify themselves as multiple-function cards. This identification allows device drivers to be created for a single subfunction independent of location, rather than a monolithic driver being created for each card.

Separate functional components that should support Plug and Play include but are not limited to:

- Broadcast receiver card
- *Microsoft Broadcast Data Network* (MSBDN) decryptor
- Video display
- MPEG decoder
- Analog tuner
- Infrared remote control

Detailed Plug and Play information can be found on the Internet on the File Transfer Protocol (FTP) server ftp.microsoft.com in the directory developr/drg/plug-and-play, and in CompuServe's PLUGPLAY forum.

# TV Viewer

[This is preliminary documentation and subject to change.]

TV Viewer is the user interface for Broadcast Architecture. It hosts several controls such as Microsoft® Internet Explorer and the Microsoft® ActiveX™ control for video (the Video control). TV Viewer uses these controls to present data such as live video, *Program Guide* information, and *enhancements*. You can expand TV Viewer functionality by writing custom controls that interact with TV Viewer.

For more information, see the following topics:

- About TV Viewer, which describes TV Viewer and the interfaces it exposes.
- Using TV Viewer, which explains how to use the interfaces exposed by TV Viewer.
- TV Viewer Reference, which provides detailed reference information.

# About TV Viewer

[This is preliminary documentation and subject to change.]

TV Viewer, the Broadcast Architecture user interface, hosts an instance of Internet Explorer so as to display layout pages in Hypertext Markup Language (HTML). These pages contain further controls that provide such functionality as displaying video or pulling Program Guide information from the *Guide database*.

TV Viewer provides two main interfaces that enable your application to interact with it:

- **ITVViewer**, the primary dispatch interface that exposes methods to programmatically control TV Viewer. For example, your application can use the **ITVViewer::Tune** method to tune TV Viewer to a new channel.
- **ITVControl**, a notification interface that your application can implement and register with TV Viewer in order to receive event notifications, such as when TV Viewer tunes to a new channel.

In addition, Broadcast Architecture defines interfaces for objects that wrap *episode* or *channel* records in the Guide database. If you pass an object that wraps a Guide database record to TV Viewer, that object must implement one of the following interfaces:

- **IEPGItem**, an abstract interface for an object that wraps a record in the Guide database. The **IEPGEpisode** interface inherits from **IEPGItem**.
- **IEPGEpisode**, an interface for an object that wraps an episode record in the Guide database.

As an example of when your application needs to implement these interfaces, suppose you want to set a *show reminder* using the **ITVViewer::SetReminder** method. **SetReminder** takes an episode object as an input parameter. In this case, you must define, create, and initialize an episode object that implements the **IEPGEpisode** interface before you pass that episode object to **SetReminder**.

## Using TV Viewer

[This is preliminary documentation and subject to change.]

You can take advantage of the TV Viewer technology by writing custom World Wide Web controls that are displayed by and interact with TV Viewer. You can also write a stand-alone application that either controls or monitors TV Viewer, or both.

If your application monitors TV Viewer, you must ensure that an instance of TV Viewer is running when your application starts. You must do so because your application cannot create a new instance of TV Viewer but instead must obtain a reference to a running instance. For more information on how to do this, see Getting a Pointer to TV Viewer.

Once your application has connected to TV Viewer, it can directly control TV Viewer using the methods of the **ITVViewer** interface. Your application can also register a sink for the **ITVControl** interface that enables your application to receive event notifications, such as when TV Viewer tunes to a new channel..

For details about working with TV Viewer, see the following topics:

- Tuning TV Viewer
- Using **ITVViewer** to Schedule a Show Reminder
- Changing the TV Viewer Display Mode
- Registering an **ITVControl** Sink
- Running TV Viewer from the command line

The Broadcast Architecture material includes a sample MFC application, Tvxsamp.exe, which demonstrates how to connect to and control TV Viewer. To locate Tvxsamp.exe, see Broadcast Architecture Sample Applications.

For additional information on the tasks involved in using TV Viewer with custom controls, see Creating TV Viewer Controls.

## Getting a Pointer to TV Viewer

[This is preliminary documentation and subject to change.]

Before your application can register a **ITVControl** notification sink or use the methods exposed by **ITVViewer**, it must first obtain a reference to the TV Viewer object. Your application cannot create a new instance of TV Viewer, so instead it must get a reference to a running instance by calling the **GetActiveObject** Automation function. To locate more information on **GetActiveObject**, see Further Information on Television Services for the Client.

Typically, your application gets this reference to TV Viewer during its initialization. Getting a TV Viewer reference at initialization ensures that TV Viewer is available before your application attempts to use it.

The following code obtains a reference to TV Viewer:

```
IUnknown *punk = NULL;
ITVViewer *ptvx = NULL;

//Get a reference to TV Viewer
GetActiveObject(CLSID_TVViewer, NULL, &punk);

// Check whether a reference to TV Viewer was returned
if (punk != NULL)
{
  'Get a reference to the ITVViewer interface
  punk->QueryInterface(IID_ITVViewer, (void **)&ptvx);

  'Release the TV Viewer object
  punk->Release();
}
```

## Tuning TV Viewer

[This is preliminary documentation and subject to change.]

Once your application has a reference to TV Viewer, as discussed in Getting a Pointer to TV Viewer, it can call the **ITVViewer::Tune** method to cause TV Viewer to display a new channel.

The following line of code tunes TV Viewer to the specified channel and tuning space. In this example, the audio and video subchannel values are each specified as –1, which causes TV Viewer to use default values.

```
TVX->Tune(iTuningSpace, iChannel,-1,-1,NULL);
```

The **ITVViewer** interface offers two other tuning-related methods, **ITVViewer::GetCurrentTuningInfo** and **ITVViewer::GetPreviousTuningInfo**. **GetCurrentTuningInfo** returns information about the channel TV Viewer is currently tuned to, and **GetPreviousTuningInfo** returns information about the channel tuned to just previously.

The following example uses **GetPreviousTuningInfo** and **Tune** methods to implement a back button, which when clicked tunes TV Viewer to the previous channel.

```
void CMyApp::BackButton()
{
  long lTuningSpacePrev;
  long lChannelNumberPrev;
  long lAudioStreamPrev;
  long lVideoStreamPrev;
  CComBSTR bstrIPAddressPrev;

  TVX->GetPreviousTuningInfo(&lTuningSpacePrev, &lChannelNumberPrev,
            &lVideoStreamPrev, &lAudioStreamPrev, &bstrIPAddressPrev);

  TVX->Tune(lTuningSpacePrev, lChannelNumberPrev,
            lVideoStreamPrev, lAudioStreamPrev, bstrIPAddressPrev);
}
```

## Using ITVViewer to Schedule a Show Reminder

[This is preliminary documentation and subject to change.]

Your application can create a *show reminder* in the Task Scheduler in the Microsoft® Windows® 98 operating system by calling the **ITVViewer::SetReminder** method. Calling **SetReminder** provides your application the same functionality that TV Viewer uses to set a show reminder.

There are three advantages to calling **SetReminder**, instead of using the alternative procedure described in Using **IScheduledItems** to Schedule a Show Reminder:

- **SetReminder** causes TV Viewer to display a **Set Reminder** dialog box to the user. The user can use this dialog to edit the reminder, for example to make it run daily or weekly, change the start time, change it remind to record, and so on.
- Reminders set using **SetReminder** can be viewed and administered from the TV Viewer user interface. In contrast, reminders set using **IScheduledItems** cannot be viewed in TV Viewer unless they meet the standards specified in Setting a Reminder that Appears in TV Viewer.
- **SetReminder** automatically builds the show reference and adds the correct path for TV Viewer to the scheduled task. If you use **IScheduledItems**, your application must compute these values.

**To set a show reminder using ITVViewer**

1. Obtain a pointer to an **ITVViewer** interface by getting a reference to an active instance of **TV Viewer**.
2. Create an **EPGEpisode** object and set its values. For more information on doing so, see the **IEPGEpisode** interface topic.
3. Call the **ITVViewer::SetReminder** method to set a reminder in the Task Scheduler.

Your application can delete a show reminder by using the **ITVViewer::DeleteReminder** method. For more information on the tasks involved in creating show reminders, see Scheduling Show Reminders.

## Changing the TV Viewer Display Mode

[This is preliminary documentation and subject to change.]

TV Viewer has two display modes, television and desktop. Television mode displays TV Viewer full-screen. Desktop mode displays TV Viewer as an application in a window.

Once your application has connected to TV Viewer, as described in Getting a Pointer to TV Viewer, you can call **ITVViewer::SetTVMode** to change the display mode from desktop to television or from television to desktop. The other display-related method TV Viewer provides is **ITVViewer::IsTVMode**, which returns a Boolean value indicating whether TV Viewer is currently displaying full-screen.

The following example combines these methods to implement an event handler for a button that toggles TV Viewer between full-screen and desktop display.

```
void CTVXSamp::ClickToggleModeButton()
{
  /* Check whether TV Viewer is currently displaying full-screen */
  if (TVX->IsTVMode())
  {
    /* If so, change to desktop mode */
    TVX->SetTVMode(false);
  }else{
    /* Otherwise, set TV Viewer to full-screen mode */
    TVX->SetTVMode(true);
  }
}
```

## Registering an ITVControl Sink

[This is preliminary documentation and subject to change.]

To receive event notifications from TV Viewer, your application must first implement the **ITVControl** interface. Then, when your application runs, it must obtain a reference to TV Viewer, as described in Getting a Pointer to TV Viewer. After obtaining this reference, your application then registers its implementation of **ITVControl** as a notification sink.

**Note** Currently, TV Viewer only sends notifications to applications running in the same process as TV Viewer. An example of such an application is an ActiveX control or component called from an enhancement page that is currently being displayed by TV Viewer.

This process is demonstrated in the following example. Note that `m_xTVControl` is a member variable implementation of **ITVControl**.

```
IUnknown *punk = NULL;
ITVViewer *ptvx = NULL;
HRESULT hr;
LPCONNECTIONPOINTCONTAINER pcpc = NULL;

//Get the ITVViewer interface
GetActiveObject(CLSID_TVViewer, NULL, &punk);

//If the interface is not found, return VARIANT_FALSE
if (punk == NULL)
  return VARIANT_FALSE;

punk->QueryInterface(IID_ITVViewer, (void **)&pvtx);
punk->Release();

//Get the IID_TVControl connection point
ptvx->QueryInterface(IID_IConnectionPointContainer, (void**)&pcpc);
if (pcpc == NULL)
{
  ptvx->Release();
  ptvx = NULL;
  return VARIANT_FALSE;
}

pcpc->FindConnectionPoint(IID_ITVControl, &pcpTVControl);
pcpc->Release();
if (pcpTVControl == NULL)
{
  ptvx->Release();
  ptvx = NULL;
  return VARIANT_FALSE;
}

//Ask the connection point to advise on m_xTVControl
hr = pcpTVControl->Advise(&m_xTVControl, &dwTVControl);
if (FAILED(hr))
{
  pcpTVControl->Release();
  pcpTVControl = NULL;
  ptvx->Release();
  ptvx = NULL;
  return VARIANT_FALSE;
}

ptvx->Release();
return VARIANT_TRUE;
```

## Running TV Viewer From the Command Line

[This is preliminary documentation and subject to change.]

Typically you do not run TV Viewer from the command line. Instead you use the user interface built into TV Viewer and Windows 98. However, TV Viewer does support command-line arguments. You can use this syntax to run TV Viewer from the command-line or to create applications that control TV Viewer by sending command lines to the shell.

For more information, see the following topics:

- Starting TV Viewer from the Command Line
- Tuning TV Viewer from the Command Line
- Toggling the Display Mode from the Command Line
- Displaying the Program Guide from the Command Line
- Displaying a Reminder from the Command Line
- Displaying a Record Reminder from the Command Line

## Starting TV Viewer from the Command Line

[This is preliminary documentation and subject to change.]

The following command line syntax starts TV Viewer. If TV Viewer is already running, this command does nothing.

You can also specify the channel that TV Viewer tunes to when it starts, for more information see, Tuning TV Viewer from the Command Line.

*<path>* **Tvx.exe**

Where

*<path>*

The path to the TV Viewer directory, typically "C:\Program Files\TV Viewer". The location of this directory is stored in the **ProductDir** value under this registry key:

**HKLM\Software\Microsoft\TV Services**

If you are running the command line from the TV Viewer directory, you do not need to specify the path.

### Examples

Both of the following command lines start TV Viewer. Note that the second example assumes that you are running the command line from the TV Viewer directory.

```
C:\Program Files\TV Viewer\Tvx.exe
Tvx.exe
```

## Tuning TV Viewer from the Command Line

[This is preliminary documentation and subject to change.]

The following command line syntax tunes TV Viewer to the specified channel. If TV Viewer is not currently running this command line also starts TV Viewer.

*<path>***Tvx.exe tv:[//]** *<tuning_identifier>*

The double back slash (//) before the tuning identifier is optional.

Where

*<path>*
The path to the TV Viewer directory, typically "C:\Program Files\TV Viewer". The location of this directory is stored in the **ProductDir** value under this registry key:

**HKLM\Software\Microsoft\TV Services**

If you are running the command line from the TV Viewer directory, you do not need to specify the path.

*<tuning_identifer>*
Specifies the channel or station to which TV Viewer should tune. This can be specified either by the channel number, the network call letters, or the station call letters.

### Examples

The following three command lines all tune TV Viewer to channel 32. Note that the last two examples assume that you are running the command line from the TV Viewer directory.

```
C:\Program Files\TV Viewer\Tvx.exe tv:32
Tvx.exe tv:32
Tvx.exe tv://32
```

The next two command lines tune TV Viewer to the local station, KCTS. The channel number associated with that station will vary in different broadcast areas.

```
Tvx.exe tv:KCTS
Tvx.exe tv://KCTS
```

The next two command lines tune TV Viewer to the national network station, NBC. The channel number associated with that station will vary in different broadcast areas.

```
Tvx.exe tv:NBC
Tvx.exe tv://NBC
```

## Displaying the Program Guide from the Command Line

[This is preliminary documentation and subject to change.]

The following command line syntax causes TV Viewer to display the program guide. If TV Viewer is not currently running this command line also starts TV Viewer.

*<path>***Tvx.exe -g**

Where

*<path>*
: The path to the TV Viewer directory, typically "C:\Program Files\TV Viewer". The location of this directory is stored in the **ProductDir** value under this registry key:

  **HKLM\Software\Microsoft\TV Services**

  If you are running the command line from the TV Viewer directory, you do not need to specify the path.

**Examples**

The following line causes TV Viewer to display the program guide.

```
C:\Program Files\TV Viewer\Tvx.exe -g
```

## Toggling the Display Mode from the Command Line

[This is preliminary documentation and subject to change.]

The following command line syntax toggles TV Viewer between windowed and full-screen display. If TV Viewer is not currently running this command line also starts TV Viewer.

*<path>***Tvx.exe -t**

Where

*<path>*
: The path to the TV Viewer directory, typically "C:\Program Files\TV Viewer". The location of

this directory is stored in the **ProductDir** value under this registry key:

**HKLM\Software\Microsoft\TV Services**

If you are running the command line from the TV Viewer directory, you do not need to specify the path.

**Examples**

The following line toggles the TV Viewer display mode. If TV Viewer is displaying as a windowed application, the command causes it to display full-screen, and vice versa.

```
C:\Program Files\TV Viewer\Tvx.exe -t
```

## Displaying a Reminder from the Command Line

[This is preliminary documentation and subject to change.]

The following command line syntax causes TV Viewer to display a show reminder dialog box for the specified episode. If TV Viewer is not currently running this command starts TV Viewer.

*<path>***Tvx.exe /b "***<show_reference>***" /u "***<user_name>***" /a "tvviewer!Remind!***<duration>***!"**

Where

<path>
The path to the TV Viewer directory, typically "C:\Program Files\TV Viewer". The location of this directory is stored in the **ProductDir** value under this registry key:

**HKLM\Software\Microsoft\TV Services**

If you are running the command line from the TV Viewer directory, you do not need to specify the path.

*<show_reference>*
The show reference of the episode for which the reminder is set. For more information, see Show Reference Format. If the show reference is invalid, or the date specified in the show reference string is not the current date, TV Viewer does not display the reminder. If the show reference is valid, but is missing some pieces of information, TV Viewer attempts to fill in the missing information by matching the show reference to an episode in the Guide database.

*<user_name>*
The name of the user setting the reminder. For version 1.0 of Broadcast Architecture this is typically "GuestUser".

*<duration>*
The duration of the show, in minutes.

**Note** The Task Scheduler uses this command-line syntax when it calls TV Viewer to display a show reminder. For more information, see Show Reminder Format. To locate more information about the Task Scheduler feature of Windows 98, see Further General Information.

**Examples**

The following command line causes TV Viewer to display a show reminder for the show titled *Inside Monster Jam.* Note that the syntax below only works if the current date is 11/4/1997.

```
C:\Program Files\TV Viewer\Tvx.exe /b "1997/11/4!0/0/0!23:30!
0!0!0!0!0!0!0!0!''!'ESN2'!'Cable'!15!Inside Monster Jam"
/u "GuestUser" /a "tvviewer!Remind!30!"
```

## Displaying a Record Reminder from the Command Line

[This is preliminary documentation and subject to change.]

The following command line syntax causes TV Viewer to display a record reminder dialog box for the specified episode and to tune to the specified channel when the episode starts. If TV Viewer is not currently running, this command starts TV Viewer.

*<path>***Tvx.exe /b "***<show_reference>***" /u "***<user_name>***" /a "tvviewer!Record!***<duration>***!"**

Where

<path>
The path to the TV Viewer directory, typically "C:\Program Files\TV Viewer". The location of this directory is stored in the **ProductDir** value under this registry key:

**HKLM\Software\Microsoft\TV Services**

If you are running the command line from the TV Viewer directory, you do not need to specify the path.

*<show_reference>*
The show reference of the episode for which the reminder is set. For more information, see Show Reference Format. If the show reference is invalid, or the date specified in the show reference string is not the current date, TV Viewer does not display the reminder. If the show reference is valid, but is missing some pieces of information, TV Viewer attempts to fill in the missing information by matching the show reference to an episode in the Guide database.

*<user_name>*
The name of the user setting the reminder. For version 1.0 of Broadcast Architecture this is typically "GuestUser".

*<duration>*
The duration of the show, in minutes.

**Note** The Task Scheduler uses this command-line syntax when it calls TV Viewer to display a record reminder. For more information, see Show Reminder Format. To locate more information about the Task Scheduler feature of Windows 98, see Further General Information.

**Examples**

The following command line causes TV Viewer to display a record reminder for the show titled "Inside Monster Jam". Note that the syntax below will only work if the current date is 11/4/1997.

```
C:\Program Files\TV Viewer\Tvx.exe /b "1997/11/4!0/0/0!23:30!
0!0!0!0!0!0!0!0!''!'ESN2'!'Cable'!15!Inside Monster Jam"
/u "GuestUser" /a "tvviewer!Record!30!"
```

## TV Viewer Reference

[This is preliminary documentation and subject to change.]

TV Viewer provides two dispatch interfaces, using which your application can interact with TV Viewer. Declared in Tvdisp.odl, these interfaces are:

- TV Viewer Registry Entries, lists the registry entries used by TV Viewer.
- **ITVViewer**, the primary dispatch interface that exposes methods that enable you to programmatically control TV Viewer. For example, using the **ITVViewer::Tune** method you can tune TV Viewer to a new channel.
- **ITVControl**, a notification interface that you can implement in your control and register with TV Viewer in order to receive event notifications.

In addition, Broadcast Architecture defines the following interfaces that wrap items in the Guide database. Your application is responsible for declaring and implementing these interfaces in objects that it creates and passes to TV Viewer. These interfaces are:

- **IEPGItem**, which is an abstract interface definition for an object that wraps Guide database information.
- **IEPGEpisode**, which is an interface for an object that wraps the field data of an episode in the Guide database.

## TV Viewer Registry Entries

[This is preliminary documentation and subject to change.]

TV Viewer uses the following registry entries to store information. These values are stored in subkeys of this registry key:

**HKLM\Software\Microsoft\TV Services\Explorer\**

| Name | Datatype | Description |
|---|---|---|
| **StartRecordingApp** | String | Path and filename of an application. TV Viewer starts this application when recording starts. For a description of the command-line send to this application, see Setting a Record Reminder. |
| **EndRecordingApp** | String | Path and filename of an application. TV Viewer starts this application when recording ends. For a description of the command-line send to this application, see Setting a Record Reminder. |
| **ClosedCaption** | DWORD | Indicates whether closed captioning is enabled. If this value is 1, TV Viewer displays close captions. If this value is 0, it does not. |
| **DistanceViewing** | Binary | Indicates whether distance viewing is enabled. If this value is 0, TV Viewer is configured for viewing on a desktop system. If this value is 1, TV Viewer is configured for distance viewing.<br><br>When TV Viewer is configured for distance viewing, another registry entry, NTSC is added. |
| **Enhancements** | DWORD | Indicates whether enhancements are enabled. If this value is 1, TV Viewer displays enhancements. If this value is 0, it does not. |
| **NTSC** | Binary | Indicates the viewing platform used for distance viewing. If this value is 1, TV Viewer is configured for a standard television. If this value is 0, TV Viewer is configured for a large screen monitor.<br><br>This value is not used if |

|  |  |  |
|---|---|---|
|  |  | **DistanceViewing** is zero. |
| **OutputDevice** | String | String value that indicates the name of an output device. If this value is present TV Viewer attempts to find the device and set full-screen video output to it.<br><br>This video output is in addition to TV Viewer's displaying video on the screen. This registry value can be used to set video output to a device such as a VCR. For example, if a VCR is installed with a device name of "AuxOut", setting OutputDevice to "AuxOut" would cause TV Viewer to send video output to the VCR. |

## ITVViewer

[This is preliminary documentation and subject to change.]

The **ITVViewer** interface provides methods that your application can use to programmatically control an instance of TV Viewer.

### When to Implement

You do not need to implement this interface. It is implemented by TV Viewer as the primary dispatch interface.

### When to Use

Your application can use this interface to call methods that programmatically control an instance of TV Viewer.

### Methods in Vtable Order

| **IUnknown Methods** | **Description** |
|---|---|
| **QueryInterface** | Returns pointers to supported interfaces |
| **AddRef** | Increments reference count |
| **Release** | Decrements reference count |

| ITVViewer | Description |
|---|---|
| **SetTVMode** | Sets TV Viewer to either television or desktop display mode. |
| **IsTVMode** | Returns a value indicating the display mode that TV Viewer is currently using. |
| **IsChannelBarUp** | Returns a value indicating whether the TV banner is currently displayed. |
| **IsModalDialogUp** | Returns a value indicating whether a modal dialog box is currently displayed. |
| **IsLoaderActive** | Returns a value indicating whether a Guide database loader is running. |
| **GlobalStartTime** | Returns the earliest start time of any episode listed in the Guide database. |
| **GlobalEndTime** | Returns latest end time of any episode listed in the Guide database. |
| **WantKeys** | Sets whether TV Viewer control traps keystrokes or passes them through to the focus window. |
| **Tune** | Tunes TV Viewer to the specified channel and *tuning space*. |
| **GetCurrentTuningInfo** | Returns information about the channel to which TV Viewer is currently tuned. |
| **GetPreviousTuningInfo** | Returns information about the last channel to which TV Viewer was tuned. |
| **SetReminder** | Sets a show reminder in the Task Scheduler for the specified episode. |
| **HasReminder** | Returns a value that indicates whether a reminder is set for the specified episode. |
| **DeleteReminder** | Deletes a reminder for the specified episode. |
| **HasEnhancement** | Returns a value that indicates whether the specified episode has enhancements. |
| **IsCC** | Returns a value that indicates whether the broadcast has closed captioning. |

### Remarks

Because TV Viewer does not support the **IClassFactory** COM interface, you cannot create a new instance of TV Viewer. Instead, you must acquire a pointer to a running instance. For more information on how to do this, see Getting a Pointer to TV Viewer.

**ITVViewer** is derived from the **IDispatch** interface. To locate more information on **IDispatch**, a Component Object Model (COM) interface, see Further Information on Television Services for the Client.

**ITVViewer** also contains the method, **ViewerID**, which is reserved for future use.

**See Also**

**ITVControl**

# ITVViewer::DeleteReminder

[This is preliminary documentation and subject to change.]

The **DeleteReminder** method deletes either a record reminder or all reminders for the specified episode, or for recurring episodes.

```
HRESULT DeleteReminder(
  IUnknown *pEpisode,
  VARIANT_BOOL bRecord
);
```

**Parameters**

*pEpisode*
Pointer to the **IEPGEpisode** interface for the episode.

*bRecord*
Boolean value that specifies whether to delete a record reminder or all reminders for the specified episode. This parameter can be one of the following values.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | Delete the record reminder. Each episode can have only one record reminder set. |
| VARIANT_FALSE | Delete all reminders. In other words, delete the record reminder and the regular reminder. |

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

**Remarks**

If the user has set a reminder for a weekly television broadcast, for example, such for as Star Trek: Voyager, this method deletes all of the reminders. If overlapping reminders exist, this method deletes only the first reminder for the specified episode.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

# ITVViewer::GetCurrentTuningInfo

[This is preliminary documentation and subject to change.]

The **GetCurrentTuningInfo** method returns information about the channel to which TV Viewer is currently tuned.

```
HRESULT GetCurrentTuningInfo(
  long *lTuningSpace,
  long *lChannelNumber,
  long *IVideoStream,
  long *lAudioStream,
  BSTR *pbsIPAddress
);
```

**Parameters**

*lTuningSpace*
: Pointer to a **long** that receives the tuning space identifier of the current channel.

*lChannelNumber*
: Pointer to a **long** that receives the current channel number.

*IVideoStream*
: Pointer to a **long** that receives the identifier of the current video stream.

*lAudioStream*
: Pointer to a **long** that receives the identifier of the current audio stream.

*pbsIPAddress*
: Pointer to a **BSTR** that receives the current IP address TV Viewer is monitoring for triggers. This address should be in the format *xxx.xxx.xxx.xxx***t***yyy*, where *xxx.xxx.xxx.xxx* specifies the IP address, and *yyy* specifies the port. The IP address and the port values are separated by a tab character, for example 255.255.255.255\t999.

  This parameter must be initialized to NULL when you pass it; otherwise, the method returns an OLE exception.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

## ITVViewer::GetPreviousTuningInfo

[This is preliminary documentation and subject to change.]

The **GetPreviousTuningInfo** method returns information about the channel to which TV Viewer was previously tuned.

```
HRESULT GetPreviousTuningInfo(
  long *lTuningSpace,
  long *lChannelNumber,
  long *IVideoStream,
  long *lAudioStream,
  BSTR *psbIPAddress
);
```

### Parameters

*lTuningSpace*
: Pointer to a **long** that receives the tuning space identifier of the previous channel.

*lChannelNumber*
: Pointer to a **long** that receive the channel number of the previous channel.

*IVideoStream*
: Pointer to a **long** that receives the identifier of the previous video stream.

*lAudioStream*
: Pointer to a **long** that receives the identifier of the previous audio stream.

*psbIPAddress*
: Pointer to a **BSTR** that receives the IP address on which triggers were sent for the previous channel. This address should in the format *xxx.xxx.xxx.xxx***t***yyy*, where *xxx.xxx.xxx.xxx* specifies the IP address, and *yyy* specifies the port. The IP address and the port values are separated by a tab character, for example 255.255.255.255\t999.

  This parameter must be initialized to NULL when you pass it to the method; otherwise. the method returns an OLE exception.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### Remarks

This method only returns information about the single channel tuned to just previous to the current channel. It cannot be used to retrieve information about channels tuned to before that.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

## ITVViewer::GlobalEndTime

[This is preliminary documentation and subject to change.]

The **GlobalEndTime** method retrieves the date and time of the latest end time of any episode in the Guide database.

```
HRESULT GlobalEndTime(
  DATE *pdate
);
```

### Parameters

*pdate*
Pointer to a **DATE** structure that receives the latest end time.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.

**Header:** Declared in tvdisp.odl.

# ITVViewer::GlobalStartTime

[This is preliminary documentation and subject to change.]

The **GlobalStartTime** method retrieves the date and time of the earliest start time of any episode in the Guide database.

```
HRESULT GlobalStartTime(
  DATE *pdate
);
```

### Parameters

*pdate*
Pointer to a **DATE** structure to receive the global start time.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

# ITVViewer::HasEnhancement

[This is preliminary documentation and subject to change.]

The **HasEnhancement** method checks the specified episode to see whether it is enhanced.

```
HRESULT HasEnhancement(
  IUnknown *pEpisode
  VARIANT_BOOL *bEnhanced
);
```

**Parameters**

*pEpisode*
Pointer to the **IEPGEpisode** interface of the episode.

*bEnhanced*
Pointer to a boolean variable that receives the enhancement information. This can be one of the following values.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | The episode is enhanced. |
| VARIANT_FALSE | The episode is not enhanced. |

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

# ITVViewer::HasReminder

[This is preliminary documentation and subject to change.]

The **HasReminder** method returns a value specifying whether a reminder is set for the specified episode.

```
HRESULT HasReminder(
  IUnknown *pEpisode,
  VARIANT_BOOL bRecord
  VARIANT_BOOL *bReminder
);
```

**Parameters**

*pEpisode*
Pointer to the **IEPGEpisode** interface of the episode. Your application must implement an episode object that supports **IEPGEpisode**. Your application initializes the object's properties with the episode data.

*bRecord*

Boolean value that indicates the type of reminder to look for. This can be one of the following values.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | A reminder to record the show. |
| VARIANT_FALSE | A reminder to watch the show. |

*bReminder*

Pointer to a boolean variable that receives the reminder information. This can be one of the following values.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | A reminder is set for the episode. |
| VARIANT_FALSE | No reminder is set for the episode. |

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

## ITVViewer::IsCC

[This is preliminary documentation and subject to change.]

The **IsCC** method returns a value that indicates closed captioning is enabled.

```
HRESULT HasEnhancement(
  VARIANT_BOOL *pbCC
);
```

### Parameters

*pbCC*

Pointer to a boolean variable that receives the closed captioning information. This can be one of the following values.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | Closed captioning is enabled.. |
| VARIANT_FALSE | Closed captioning is disabled. |

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

# ITVViewer::IsChannelBarUp

[This is preliminary documentation and subject to change.]

The **IsChannelBarUp** method returns a value that specifies whether the TV Viewer menu bar is currently displayed.

```
HRESULT IsChannelBarUp(
  VARIANT_BOOL *pfBarUp
);
```

**Parameters**

*pfBarUp*
Pointer to a boolean variable that receives the TV banner information. This can be one of the following values.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | The TV banner is currently displayed. |
| VARIANT_FALSE | The TV banner is hidden. |

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

# ITVViewer::IsLoaderActive

[This is preliminary documentation and subject to change.]

The **IsLoaderActive** method returns a value indicating whether a Guide database loader application is running.

```
HRESULT IsLoaderActive(
  VARIANT_BOOL *pfLoaderActive
);
```

### Parameters

*pfLoaderActive*
Pointer to a boolean variable that receives the loader activity information. This can be one of the following values.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | A Guide database loader is running. |
| VARIANT_FALSE | No Guide database loader components are running. |

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

# ITVViewer::IsModalDialogUp

[This is preliminary documentation and subject to change.]

The **IsModalDialogUp** method returns a value specifying whether a modal dialog box is currently displayed.

```
HRESULT IsModalDialogUp(
  VARIANT_BOOL *pfModalUp
);
```

### Parameters

*pfModalUp*

Pointer to a boolean variable that receives the information on dialog box display. This can be one of the following values.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | A modal dialog box is currently displayed. |
| VARIANT_FALSE | No modal dialog boxes are currently displayed. |

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

# ITVViewer::IsTVMode

[This is preliminary documentation and subject to change.]

The **IsTVMode** method returns a value specifying whether TV Viewer currently displays in television or desktop mode.

```
HRESULT IsTVMode(
  VARIANT_BOOL *pfTVmode
);
```

### Parameters

*pfTVmode*

Pointer to a boolean variable that receives the television mode information. This can be one of the following values.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | TV Viewer is currently displaying in TV, or full-screen, mode. |
| VARIANT_FALSE | TV Viewer is currently displayed in desktop, or window, mode. |

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

## ITVViewer::SetReminder

[This is preliminary documentation and subject to change.]

The **SetReminder** method sets a reminder for the specified episode. The reminder is set as a task in the Task Scheduler.

```
HRESULT SetReminder(
  IUnknown *pEpisode,
  VARIANT_BOOL bRecord
);
```

### Parameters

*pEpisode*

Pointer to an **IEPGEpisode** interface. Your application must implement an episode object that supports **IEPGEpisode**. Your application initializes the object's properties with data that reflects the episode to schedule.

*bRecord*

Boolean value that specifies whether the reminder should be a record reminder. This can be one of the following values.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | Set a reminder to record a broadcast. |
| VARIANT_FALSE | Set a reminder to watch a broadcast. |

### Remarks

If the reminder is a record reminder, you should use the Task Scheduler to set the TASK_FLAG_SYSTEM_REQUIRED flag for the reminder. This causes TV Viewer to tune to the channel even if the system is sleeping. Otherwise, if the system is sleeping, TV Viewer will not wake up to run the record reminder.

In addition, if the record reminder has an application associated with it that automates tuning the VCR this application should be specified in the **StartRecordingApp** and/or **EndRecordingApp** values under this registry key:

**HKLM\Software\Microsoft\TV Services\Explorer\**

The TASK_FLAG_SYSTEM_REQUIRED flag should not be set for standard show reminders. Version 1.0 of Broadcast Architecture does not handle show reminders that go off while the system is sleeping.

For more information see Setting a Show Reminder and Setting a Record Reminder.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

## ITVViewer::SetTVMode

[This is preliminary documentation and subject to change.]

The **SetTVMode** method displays TV Viewer in the specified mode, either television or desktop.

```
HRESULT SetTVMode(
  VARIANT_BOOL fTVMode
);
```

### Parameters

*fTVMode*

Boolean value that indicates the mode. This can be one of the following values.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | Television mode. The control displays full-screen. |
| VARIANT_FALSE | Desktop mode. The control displays in a desktop window. |

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

# ITVViewer::Tune

[This is preliminary documentation and subject to change.]

The **Tune** method tunes the TV Viewer display to the specified channel.

```
HRESULT Tune(
  long lTuningSpace,
  long lChannelNumber,
  long lVideoStream,
  long lAudioStream,
  BSTR bsIPStream
);
```

**Parameters**

*lTuningSpace*
: Tuning space of the input. Typically, each input device uses a separate tuning space.

*lChannelNumber*
: Channel number.

*lVideoStream*
: Video input stream. If you pass – 1, TV Viewer uses the default video stream as defined by the broadcast content provider. To access alternate video streams, pass the video stream identifier specified by the content provider.

*lAudioStream*
: Audio input stream. A single video input stream may have several audio streams associated with it. If you pass in a value of – 1, TV Viewer uses the default audio stream, as defined by the broadcast content provider.

  This functionality is useful because, for example, a television station with a Hispanic audience might define a Spanish-language audio stream as the default but also define an English-language version as an alternate audio stream. To access the English-language version in this case, an application passes the audio stream identifier that the content provider specifies for that version.

*bsIPStream*
: String that contains information about the IP stream used by triggers. This string must be in the format "*xxx.xxx.xxx.xxx yyy.yyy.yyy.yyy***:***zzz preloadURL***&***overlayCSS*", where *xxx.xxx.xxx.xxx* specifies the netcard address, *yyy.yyy.yyy.yyy* specifies the announcement IP address, zzz specifies the announcement port, *preloadURL* specifies the preload URL and *overlayCSS* specifies the overlay style sheet. For example: "255.255.255.255 123.25.433.1:1701 basepage.htm&basestyle.css"

  The *preloadURL* and *overlayCSS* parameters are optional, and can be left out of the string. For example, "255.255.255.255 123.25.433.1:1701".

  If no trigger stream exists, this value is set to NULL.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

# ITVViewer::WantKeys

[This is preliminary documentation and subject to change.]

The **WantKeys** method sets whether TV Viewer traps keystrokes or passes them to the window with focus.

```
HRESULT WantKeys(
  int nKeys
);
```

### Parameters

*nKeys*

A flag that specifies which type of key events TV Viewer passes to the focus window. This can be a combination of the following values.

| Value | Meaning |
|---|---|
| keNumKeys | TV Viewer passes number key (0-9) events to the focus window. |
| kePageKeys | TV Viewer passes page key (PageUp/PageDown) events to the focus window. |
| keNoKeys | TV Viewer traps both page and number key events. |
| keChannelKeys | TV Viewer passes channel key events to the focus window. |

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### Remarks

The default behavior is for TV Viewer to trap number keys and use them as input to change the channels.

The *nKeys* flags may be ORed together. For example, the following call to **WantKeys** causes TV Viewer to pass both number keys and PageUp/PageDown keys to the focus window.

```
WantKeys ( keNumKeys | kePageKeys );
```

Because **WantKeys** causes TV Viewer to send key events to the focus window, a control should ensure that it has focus before it calls this method. Otherwise, the key events will be sent to the focus application, not the control, possibly resulting in unpredictable behavior.

TV Viewer's behavior, whether it traps or forwards key events, is set by the most recent call to **WantKeys**. Furthermore, the key event behavior is set for all controls on the broadcast client. In other words, you cannot cause TV Viewer to forward key events to some controls and not others. For example, if control control A calls **WantKeys**, setting *nKeys* to keNumKeys, and then control B calls **WantKeys**, setting *nKeys* to keNoKeys, TV Viewer will use the most recent key event setting, keNoKeys, and neither control will receive key events.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

## ITVControl

[This is preliminary documentation and subject to change.]

The **ITVControl** interface is a sink that enables objects to receive notifications of changes in the state of TV Viewer.

### When to Implement

Implement **ITVControl** if you are developing a television-aware control that resides in the TV Viewer container, or if your control needs notification of TV Viewer events.

**Note** Currently, TV Viewer only sends notifications to applications running in the same process as TV Viewer. An example of such an application is an ActiveX control or component called from an enhancement page that is currently being displayed by TV Viewer.

### When to Use

TV Viewer calls the methods of **ITVControl** to notify registered objects of a change in the TV Viewer state.

### Methods in Vtable Order

| **IUnknown Methods** | **Description** |
|---|---|
| **QueryInterface** | Returns pointers to supported interfaces |
| **AddRef** | Increments reference count |
| **Release** | Decrements reference count |

| ITVControl | Description |
|---|---|
| **OnIdle** | Advises that idle-time processing is available. |
| **Tune** | Advises that TV Viewer has tuned to a new channel. |
| **TearDown** | Advises that TV Viewer is closing the current World Wide Web page. |
| **SyncEvent** | Advises that a sync event has occurred. |
| **EpisodeStatusChanged** | Advises that the status of an episode has changed. |
| **PowerChange** | Advises that the system power is turning on or turning off. |
| **OnTVFocus** | Advises that TV Viewer has gotten focus. |
| **SetOutput** | Advises that the control should set an additional output device, for example a VCR. |
| **GetCC** | Returns the closed-captioning status of the control. |
| **SetCC** | Advises that the closed-captioning status has changed in TV Viewer. |
| **EnableVideo** | Advises that the control that the status of the video display has changed. |

**Remarks**

After your object has registered a sink, it should advise on the **ITVControl** connection point when it is user interface–activated, able to interact with the user, and revoke the connection point when it is user interface–deactivated, or hidden. In other words, the control should advise when it receives a **IOleObject::DoVerb** method call specifying the value OLEVERB_UIACTIVATE, OLEVERB_INPLACEACTIVATE, or OLEVERB_PRIMARY, and the control should revoke the connection point when it receives a **IOleObject::DoVerb** call specifying the value OLEVERB_HIDE.

To locate more information about the **IOleObject** interface and the **DoVerb** method, see Further Information on Television Services for the Client.

**ITelevisionServices** is derived from the **IDispatch** interface. To locate more information on **IDispatch**, a Component Object Model (COM) interface, see Further Information on Television Services for the Client.

**See Also**

**ITVViewer**

# ITVControl::EnableVideo

[This is preliminary documentation and subject to change.]

The **EnableVideo** method advises that the status of the video display has changed.

```
HRESULT EnableVideo(
  VARIANT_BOOL bEnable,
  int iReason
);
```

### Parameters

*bEnable*

A boolean value that indicates whether the control should display video. If *bEnable* is VARIANT_TRUE, video should be enabled. If *bEnable* is VARIANT_FALSE, video should be disabled.

*iReason*

If *bEnable* is VARIANT_FALSE, this parameter contains a value indicating the reason that video is disabled. If *bEnable* is VARIANT_TRUE, this parameter is not used. The supported values are listed in the following table:

| Value | Description |
|---|---|
| kePrimaryMonitor | Video is not supported off of the primary monitor.<br><br>This can occur in a multiple-monitor environment when the user drags the TV Viewer window from the primary monitor to a secondary montor. Because video is not supported on secondary monitors, it is disabled. |

### Return Values

TV Viewer ignores the value returned by this method.

### Remarks

TV Viewer calls **EnableVideo** on all registered **ITVControl** sinks when the status of video display changes. For example, in a multiple-monitor environment if the user drags the TV Viewer window off of the primary monitor, video is no longer available because video is not supported on secondary monitors. In that case, TV Viewer calls `EnableVideo(FALSE, kePrimaryMonitor)` on all registered controls to inform them that video is disabled. The *iReason* parameter contains extended information explaining the reason that video was disabled. The control can then handle the event, presenting a message box to the user explaining why video was lost. When the TV Viewer window is moved back onto the primary monitor and video support is regained, TV Viewer calls `EnableVideo ( VARIANT_TRUE, 0 )` on all registered controls to advise them that video is re-enabled.

If a control registers an **ITVControl** sink while video is disabled, TV Viewer immediately sends an `EnableVideo(FALSE, iReason)` notification to the control. If the control does not receive this notification, it should assume that video is enabled. TV Viewer does not send a `EnableVideo(TRUE, 0)` message to newly registered controls. TV Viewer only calls `EnableVideo(TRUE, 0)` when video is re-enabled following a previous call to `EnableVideo(FALSE, iReason)`.

# ITVControl::EpisodeStatusChanged

[This is preliminary documentation and subject to change.]

The **EpisodeStatusChanged** method advises that the status of an episode has changed.

```
HRESULT EpisodeStatusChanged(
  int iChange,
  IUnknown *pEpisode
);
```

### Parameters

*iChange*

Change identifier. This identifier can be one of the following values.

| Value | Meaning |
|---|---|
| keReminderStatus | Specifies that a reminder has either been set or deleted for the episode. |
| kePurchaseStatus | Reserved. |
| keDSSEmailStatus | Reserved. |
| keEnhancementStatus | Specifies that the enhancement status has changed. Typically, this indicates that an episode or channel enhancement is now available. |

*pEpisode*

Pointer to the **IEPGEpisode** interface of an **EPGEpisode** object that contains information about the episode whose status has changed.

### Return Values

TV Viewer ignores the value returned by this method.

### Remarks

TV Viewer calls this method.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.
**Import Library:** user-defined.

## ITVControl::GetCC

[This is preliminary documentation and subject to change.]

The **GetCC** method retrieves the closed-captioning status of the control.

```
HRESULT GetCC(
  VARIANT_BOOL *bCC
);
```

**Parameters**

*bCC*

Pointer to a boolean variable that receives the closed-captioning status of the control. This can contain one of the following values.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | The control is in closed-captioning mode. |
| VARIANT_FALSE | The control is not in closed-captioning mode. |

**Return Values**

TV Viewer ignores the value returned by this method.

**Remarks**

TV Viewer calls this method.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.

**Header:** Declared in tvdisp.odl.
**Import Library:** user-defined.

## ITVControl::OnIdle

[This is preliminary documentation and subject to change.]

The **OnIdle** method advises that idle-time processing is available.

```
HRESULT OnIdle(
  VARIANT_BOOL *pbIdle
);
```

### Parameters

*pbIdle*

Pointer to a boolean that contains the idle information. This can contain one of the following values.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | The object needs additional idle processing time. |
| VARIANT_FALSE | The object does not additional idle processing time. |

### Return Values

TV Viewer ignores the value returned by this method.

### Remarks

TV Viewer calls **OnIdle** when it receives an on idle message from the operating system. Calling **OnIdle** enables registered controls to perform idle-time processing.

When the operating system has idle time available, it calls the **OnIdle** method for the main windows of the running applications. If an application indicates it needs more idle time by setting **pbIdle* to VARIANT_TRUE, the system calls the **OnIdle** method for that application again during the next idle loop. Otherwise, the Microsoft® Windows® operating system does not call **OnIdle** for that application until after the application processes another normal message.

When the system calls the TV Viewer implementation of **OnIdle**, it in turn calls the **ITVControl::OnIdle** method implemented by any registered sinks. If your application sets **pbIdle* to VARIANT_TRUE, indicating that it needs additional idle time processing, TV Viewer passes this value to the operating system. Passing this value causes all controls registered with TV Viewer to get another **OnIdle** call during the next system idle loop. The frequency of idle calls depends on system activity.

For more information about idle processes, see Further Information on Television Services for the Client.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.
**Import Library:** user-defined.

# ITVControl::OnTVFocus

[This is preliminary documentation and subject to change.]

The **OnTVFocus** method advises that TV Viewer has gotten focus.

```
HRESULT OnTVFocus(void);
```

**Parameters**

None.

**Return Values**

TV Viewer ignores the value returned by this method.

**Remarks**

TV Viewer calls this method when it gets focus. Your control can use this method to force focus to a specific window.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.
**Import Library:** user-defined.

# ITVControl::PowerChange

[This is preliminary documentation and subject to change.]

The **PowerChange** method advises that broadcast client is turning on or turning off.

```
HRESULT PowerChange(
  VARIANT_BOOL bPowerOn,
  VARIANT_BOOL bUIAllowed
);
```

### Parameters

*bPowerOn*

Boolean value that specifies whether the power is turning on or off, which can be one of the following.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | The system power is turning on. |
| VARIANT_FALSE | The system power is turning off. |

*bUIAllowed*

Boolean value that specifies whether the control can present a user interface to the viewer. For example, when power is shut off, a control might query the user whether it should save data to disk. This value can be one of the following.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | The control can present a user interface. |
| VARIANT_FALSE | The control cannot present a user interface. |

### Return Values

If the power is turning on, TV Viewer ignores the return value.

If the power is turning off, your control should return one of the following values.

| Value | Meaning |
|---|---|
| S_OK | The system can power off. |
| S_FALSE | The system should not power off. If the *bUIAllowed* parameter is VARIANT_FALSE, the system does not honor the control's request for power to stay on. |

### Remarks

TV Viewer calls this method when the power status of the system changes.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.
**Import Library:** user-defined.

## ITVControl::SetCC

[This is preliminary documentation and subject to change.]

The **SetCC** method advises that the closed-captioning status has changed in TV Viewer.

```
HRESULT SetCC(
  VARIANT_BOOL bCC
);
```

### Parameters

*bCC*

Boolean value that indicates the closed captioning state, which can be one of the following.

| Value | Meaning |
|---|---|
| VARIANT_TRUE | Closed captioning is turned on. |
| VARIANT_FALSE | Closed captioning is turned off. |

### Return Values

TV Viewer ignores the value returned by this method.

### Remarks

TV Viewer calls this method when the closed-captioning status of TV Viewer changes.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.

**Import Library:** user-defined.

## ITVControl::SetOutput

[This is preliminary documentation and subject to change.]

The **SetOutput** method advises that the control should set an additional output device, for example a VCR.

```
HRESULT SetOutput(
  BTR bsDeviceName
);
```

### Parameters

*bsDeviceName*
Value that specifies the name of the output device to set. TV Viewer looks in the registry for a device that matches this name.

### Return Values

TV Viewer ignores the value returned by this method.

### Remarks

TV Viewer calls this method so that the control can set an additional output device, in addition to regular video. For example, TV Viewer can call this method to request that a control set VCR output.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.
**Import Library:** user-defined.

## ITVControl::SyncEvent

[This is preliminary documentation and subject to change.]

The **SyncEvent** method advises that a synchronization event has occurred.

```
HRESULT SyncEvent(
  int iEvent,
  BSTR pParm1,
  BSTR pParm2
);
```

**Parameters**

*iEvent*

TV Viewer event that occurred, which can be any of the following.

| Event | Description |
|---|---|
| keCurrentViewerChannelListChange | A current viewer's channel lineup has changed. Controls should refresh any cached channel information. |
| EPGLDR_ACTIVE_COMMIT_ENDING | A Guide database loader has finished committing data to the database. Controls can now access the database and should refresh any cached data. |
| EPGLDR_ACTIVE_COMMIT_STARTING | A Guide database loader has begun to commit changes to the database. Controls should not access the database until the keEpgLdrActiveCommitEnding event is sent. |
| EPGLDR_ENDING | A Guide database loader has stopped running. |
| EPGLDR_PASSIVE_COMMIT_ENDING | A Guide database loader has finished committing data to the database. Controls can now access the database but do not need to refresh any cached data. |
| EPGLDR_PASSIVE_COMMIT_STARTING | A Guide database loader has begun to commit data to the database. Controls should not access the database until the keEpgLdrPassiveCommitEnding event is sent. |
| EPGLDR_STARTING | A Guide database loader has begun running. Controls can still access data in the database. |
| keSysTimeChange | The system time has been updated. |
| keViewerChange | Reserved. |
| keViewerLogin | Reserved. |

**Note** Note that the loader events, those that start with EPGLDR_ are defined in Epgldrx.h.

*pParm1*
: New viewer name, if a keViewerChange event has been sent. Other events do not use this parameter.

*pParm2*
: New viewer password, if a keViewerChange event has been sent. Other events do not use this parameter.

### Return Values

TV Viewer ignores the value returned by this method.

### Remarks

TV Viewer calls this method when a TV Viewer event occurs, as described in the table preceding.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.
**Import Library:** user-defined.

## ITVControl::TearDown

[This is preliminary documentation and subject to change.]

The **TearDown** method advises that TV Viewer is closing the current Web page.

```
HRESULT Teardown(void);
```

### Parameters

None.

### Return Values

TV Viewer ignores the value returned by this method.

### Remarks

This method is called by TV Viewer before it navigates to a new HTML page. Calling **TearDown** enables your control to clean up or persistently store its state before the Web page is closed.

TV Viewer displays Web pages using the browser component of Microsoft® Internet Explorer. TV Viewer calls **TearDown** before it calls the browser to navigate to a new Web page. Note that TV Viewer waits for registered applications to return a value. Thus, your control should return a value promptly, so as not to delay TV Viewer in navigating to the new page.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.
**Import Library:** user-defined.

## ITVControl::Tune

[This is preliminary documentation and subject to change.]

The **Tune** method advises that TV Viewer has tuned to a new channel.

```
HRESULT Tune(
  Long ltsNew,
  Long lcnNew,
  Long IvsNew,
  Long lasNew,
  BSTR bslPNew,
  Long ltsPrev,
  Long lcnPrev,
  Long IvsPrev,
  Long lasPrev,
  BSTR bslPPrev
);
```

### Parameters

*ltsNew*
    Tuning space of the new channel.
*lcnNew*
    Channel number of the new channel.
*IvsNew*
    Video subchannel of the new channel. A value of – 1 indicates the default video subchannel, as defined by the broadcast content provider.
*lasNew*

Audio subchannel of the new channel. A value of – 1 indicates the default audio subchannel, as defined by the broadcast content provider.

*bslPNew*

String that contains information about the IP stream used by triggers on the new channel. This string must be in the format "*xxx.xxx.xxx.xxx yyy.yyy.yyy.yyy*:*zzz preloadURL*&*overlayCSS*", where *xxx.xxx.xxx.xxx* specifies the netcard address, *yyy.yyy.yyy.yyy* specifies the announcement IP address, zzz specifies the announcement port, *preloadURL* specifies the preload URL and *overlayCSS* specifies the overlay style sheet. For example: "255.255.255.255 123.25.433.1:1701 basepage.htm&basestyle.css"

The *preloadURL* and *overlayCSS* parameters are optional, and can be left out of the string. For example, "255.255.255.255 123.25.433.1:1701".

If no trigger stream exists, this value is set to NULL.

*ltsPrev*

Tuning space of the previous channel.

*lcnPrev*

Channel number of the previous channel.

*IvsPrev*

Video subchannel of the previous channel. A value of – 1 indicates the default video subchannel, as defined by the broadcast content provider.

*lasPrev*

Audio subchannel of the previous channel. A value of – 1 indicates the default audio subchannel, as defined by the broadcast content provider.

*bslPPrev*

String that contains information about the IP stream used by triggers on teh previous channel. The string is formatted as described in *bsIPNew*.

If no trigger stream exists, this value is set to NULL.

**Return Values**

TV Viewer ignores the value returned by this method.

**Remarks**

TV Viewer calls this method when tuning information changes. This method is also called when an object first registers an **ITVControl** sink with TV Viewer.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tvdisp.odl.
**Import Library:** user-defined.

# IEPGItem

[This is preliminary documentation and subject to change.]

The **IEPGItem** interface provides methods to retrieve program guide data from an object.

## When to Implement

Implement **IEPGItem** in objects that wrap Guide database records. These objects are required as input parameters during calls to methods such as **ITVViewer::SetReminder**

## When to Use

You can call the methods of **IEPGItem** to retrieve field data from the object.

## Methods in Vtable Order

| IUnknown Methods | Description |
|---|---|
| **QueryInterface** | Returns pointers to supported interfaces |
| **AddRef** | Increments reference count |
| **Release** | Decrements reference count |

| IEPGItem | Description |
|---|---|
| **TuningInfo** | Gets the tuning space identifier and channel. |
| **StartTime** | Gets the starting time. |
| **EndTime** | Gets the ending time. |
| **Length** | Gets the length of the item, in minutes. |
| **OnNow** | Returns a value indicating whether the item is being broadcast now. In other words, this method indicates whether the current system time is later than the item's starting time and earlier than the item's ending time. |
| **Title** | Gets the title. |
| **BodyText** | Gets the description. |
| **PreviewGraphic** | Gets the file name of the preview graphic. |
| **NumIcons** | Returns the total number of icons. |
| **GetIcon** | Gets the file name of the specified icon. |
| **NumOptions** | Returns the total number of options. |
| **OptionPrompt** | Gets the option prompt. |
| **GetOption** | Gets the specified option. |

### Remarks

**IEPGItem** is an abstract interface is inherited by objects that wrap records stored in the Guide database. The **IEPGEpisode** interface inherits from **IEPGItem**.

### See Also

**ITVViewer::DeleteReminder**, **ITVViewer::HasEnhancement**, **ITVViewer::HasReminder**, **ITVViewer::SetReminder**, **ITVControl::EpisodeStatusChanged**

## IEPGItem::BodyText

[This is preliminary documentation and subject to change.]

The **BodyText** method gets the description of the item. Typically, the description text retrieved by **BodyText** is the as stored in the Guide database.

```
HRESULT BodyText (
  BSTR* pstrBodyText
);
```

### Parameters

*pstrBodyText*
Pointer to a **BSTR** that receives the item description.

### Remarks

The subject of the description depends on the object implementating **IEPGItem**. For example, in an episode object, one that implements **IEPGEpisode**, the **BodyText** method retreives a description of the episode.

Typically, the description text for an episode object is the same text stored in the Guide database. For an object that correponds to an episode in the Guide database, the description text is stored in the E Description field of the Episode table.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGItem::EndTime

[This is preliminary documentation and subject to change.]

The **EndTime** method gets the ending time of the item.

```
HRESULT EndTime(
  DATE* pEndTime
);
```

### Parameters

*pEndTime*
Pointer to a **DATE** structure that receives the date and time the item ends.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGItem::GetIcon

[This is preliminary documentation and subject to change.]

The **GetIcon** method retrieves the file name of the specified icon associated with the item.

```
HRESULT GetIcon (
  long iIconNumber,
  BSTR* pstrIconName
);
```

### Parameters

*iIconNumber*
: Zero-based identifier of the icon. This value cannot be greater than the value returned by the **IEPGItem::NumIcons** method – 1.

*pstrIconName*
: Pointer to a **BSTR** that receives the file name of the icon identified by the *iIconNumber* parameter.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### Remarks

Icons of the type retrieved by **GetIcon** represent additional information about the broadcast, such as its rating and pay-per-view status and whether it has closed captioning.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

## IEPGItem::GetOption

[This is preliminary documentation and subject to change.]

The **GetOption** method retrieves the specified option, or command, from the item.

```
HRESULT GetOption (
  long iOptionNumber,
  long* plID,
  BSTR* pstrText
);
```

### Parameters

*iOptionNumber*
A **long** that specifies the zero-based identifier of the option to retrieve. This value cannot be greater than **IEPGItem::NumOptions** – 1.

*plID*
Pointer to a **long** that receives the interface identifier (IID) of the option specified by the *iOptionNumber* parameter.

*pstrText*
Pointer to a **BSTR** that receives the name of the command. This name is the same text as displayed on the option buttons in the Program Guide, such as **Watch**, **Remind**, or **Other Times**.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

## IEPGItem::Length

[This is preliminary documentation and subject to change.]

The **Length** method returns the length, in minutes, of the item.

```
HRESULT Length (
  long* pLength
);
```

### Parameters

*pLength*
Pointer to a **long** that receives the length of the item, in minutes.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGItem::NumIcons

[This is preliminary documentation and subject to change.]

The **NumIcons** method counts the icons associated with the item.

```
HRESULT NumIcons (
  long* pNumIcon
);
```

### Parameters

*pNumIcon*
Pointer to a **long** that receives the total number of icons.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### Remarks

Icons of the type counted by **NumIcons** represent additional information about the broadcast, such as its rating and pay-per-view status and whether it has closed captioning.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGItem::NumOptions

[This is preliminary documentation and subject to change.]

The **NumOptions** method counts the number of commands that can run on an item.

```
HRESULT NumOptions (
  long* pNumOptions
);
```

### Parameters

*pNumOptions*
Pointer to a **long** that receives the total number of options.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### Remarks

The available options for a broadcast vary over time. For example, you can watch an episode that is on now, but you can only set a reminder for one that is on later.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGItem::OnNow

[This is preliminary documentation and subject to change.]

The **OnNow** method returns a Boolean specifying whether the item is currently being broadcast.

```
HRESULT OnNow (
  VARIANT_BOOL* pOnNow
);
```

### Parameters

*pOnNow*

Pointer to a boolean that indicates whether this item is currently being broadcast. If the received value is VARIANT_TRUE, the item is currently being broadcast. If the value is VARIANT_FALSE, it is not.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

## IEPGItem::OptionPrompt

[This is preliminary documentation and subject to change.]

The **OptionPrompt** method retrieves a string that explains the options available.

```
HRESULT OptionPrompt (
  BSTR* pstrPrompt
);
```

### Parameters

*pstrPrompt*

Pointer to a **BSTR** that receives the option prompt message.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.

**Windows CE:** Unsupported.
**Import Library:** user-defined.

## IEPGItem::PreviewGraphic

[This is preliminary documentation and subject to change.]

The **PreviewGraphic** method gets the file name of a graphic to display when the item is not currently being broadcast.

```
HRESULT PreviewGraphic (
  BSTR* pstrPreviewGraphic
);
```

### Parameters

*pstrPreviewGraphic*
Pointer to a **BSTR** that receives the file name. If no preview graphic exists, the method returns a NULL **BSTR**.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### Remarks

For example, a preview graphic could be displayed as a placeholder for live video from a channel that is currently off the air.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

## IEPGItem::StartTime

[This is preliminary documentation and subject to change.]

The **StartTime** method gets the starting time of the item.

```
HRESULT StartTime(
  DATE* pStartTime
);
```

### Parameters

*pStartTime*
Pointer to a **DATE** structure that receives the date and time the item starts.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

## IEPGItem::Title

[This is preliminary documentation and subject to change.]

The **Title** method gets the title of the item.

```
HRESULT Title (
  BSTR* pstrTitle
);
```

### Parameters

*pstrTitle*
Pointer to a **BSTR** that receives the title.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGItem::TuningInfo

[This is preliminary documentation and subject to change.]

The **TuningInfo** method gets tuning information associated with the item.

```
HRESULT TuningInfo (
  LONG* plTuningSpace,
  LONG* plChannelNumber
);
```

### Parameters

*plTuningSpace*
Pointer to a **long** that receives the tuning space identifier.
*plChannelNumber*
Pointer to a **long** that receives the channel number.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

## IEPGEpisode

[This is preliminary documentation and subject to change.]

The **IEPGEpisode** interface enables applications such as TV Viewer to retrieve data from an episode object.

**When to Implement**

Implement **IEPGEpisode** in objects that wrap episode records in the Guide database.

**When to Use**

You can call the methods of **IEPGEpisode** to retrieve episode field data from the object.

**Methods in Vtable Order**

| **IUnknown Methods** | **Description** |
|---|---|
| **QueryInterface** | Returns pointers to supported interfaces |
| **AddRef** | Increments reference count |
| **Release** | Decrements reference count |

| **IEPGEpisode** | **Description** |
|---|---|
| **TimeSlotID** | Gets the time-slot identifier. This value maps to the starting and ending time of the episode. |
| **ChannelID** | Gets the channel identifier. |
| **EpisodeID** | Gets the episode identifier. |
| **PayPerView** | Retrieves a Boolean value indicating whether the episode is pay-per-view. |
| **RatingID** | Gets the rating identifier. |
| **CallLetters** | Retrieves the call letters of the station. |
| **IsRemindItem** | Returns a value indicating whether a show reminder is set for the episode. |
| **IsRecordItem** | Returns a value indicating whether a record reminder is set for the episode. |
| **Repetition** | Counts the number of times that the episode appears in the Guide database. |
| **RemindRecordIdx** | Returns an index into the cache of show reminders in TV Viewer. This method is only valid for episodes for which the **IsRemindItem** or **IsRecordItem** method returns VARIANT_TRUE. |
| **IsContinuous** | Returns a value indicating whether the episode is continuous, in other words whether there is a single on-going episode for the channel. |

| | |
|---|---|
| **IsOnLater** | Returns a value indicating whether the episode is on later. |
| **IsOnSoon** | Returns a value indicating whether the episode is on within the next five minutes. |
| **ThemeID** | Gets the episode's theme identifier. |
| **AbbreviatedTitle** | Gets the abbreviated version of the episode title. |
| **HasEnhancement** | Gets a value indicating whether the episode is enhanced. |
| **Layout** | Gets the enhancement layout information. |

**Remarks**

The **IEPGEpisode** interface enables applications such as TV Viewer to retrieve data from an episode object. An episode object is one that wraps the data of an episode record in the Guide database. **IEPGEpisode** inherits from the **IEPGItem** interface. **IEPGItem** must be implemented in episode objects passed to TV Viewer.

TV Viewer creates its own episode objects and initializes them with data from the Guide database. For example, TV Viewer can query the Guide database to open a recordset of all the episodes that have "great" in the title. For each record in the recordset, TV Viewer creates an episode object, initializing its values to the record's field values.

Pointers to **IEPGEpisode** interfaces are passed as parameters to the **ITVViewer::SetReminder** and **ITVViewer::HasEnhancement** methods. To use these methods, your application must implement an episode object that supports **IEPGEpisode**.

TV Viewer returns an **IEPGEpisode** pointer when it calls the **ITVControl::EpisodeStatusChanged** method.

The **IEPGEpisode** interface also contains the following methods, which are reserved for future use.

- **IsListGuideItem**
- **IsPurchasable**
- **IsPurchaseItem**
- **PPVCanBeCancelled**
- **PPVTapeCost**
- **PPVTokenAddress**
- **PPVViewCost**
- **PurchaseIdx**

**See Also**

**IEPGItem**

## IEPGEpisode::AbbreviatedTitle

[This is preliminary documentation and subject to change.]

The **AbbreviatedTitle** method gets the short version of the episode title.

```
HRESULT AbbreviatedTitle(
  BSTR* pstrAbbrevTitle
);
```

### Parameters

*pstrAbbrevTitle*
Pointer to a **BSTR** that receives the abbreviated title.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### Remarks

The abbreviated title contains the content-provider's recommended abbreviation of the episode title. The abbreviated version of the title should be used in user interfaces where there is insufficient space to display the full title. For example, "The Great Adventures of Mulligan" might have an abbreviated title of "Adventures of Mulligan".

## IEPGEpisode::CallLetters

[This is preliminary documentation and subject to change.]

The **CallLetters** method gets the call letters associated with the episode. This information corresponds to the S Call Letters field in the Guide database.

```
HRESULT CallLetters(
  BSTR* ppszCallLetters
);
```

### Parameters

*ppszCallLetters*
Pointer to a **BSTR** that receives the call letters.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGEpisode::ChannelID

[This is preliminary documentation and subject to change.]

The **ChannelID** method retrieves the channel identifier of the episode. This information corresponds to the TS Channel ID field in the Guide database.

```
HRESULT ChannelID(
  long* lChannelID
);
```

### Parameters

*lChannelID*
Pointer to a **long** that receives the channel identifier.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGEpisode::EpisodeID

[This is preliminary documentation and subject to change.]

The **EpisodeID** method retrieves the identifier of the episode. This information corresponds to the TS Episode ID field in the Guide database.

```
HRESULT EpisodeID(
  long* lEpisodeID
);
```

## Parameters

*lEpisodeID*
Pointer to a **long** that receives the episode identifier.

## Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

## QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGEpisode::HasEnhancement

[This is preliminary documentation and subject to change.]

The **HasEnhancement** method gets a value indicating whether the episode is enhanced. If the episode is enhanced, this method retrieves identifiers specifying the enhancement.

```
HRESULT HasEnhancement(
  VARIANT_BOOL *pf,
  long * lTSEnhMappingID,
  long * lEpiEnhMappingID
);
```

## Parameters

*pf*
: Pointer to a boolean value that receives VARIANT_TRUE if the episode is enhanced, or VARIANT_FALSE otherwise.

*lTSEnhMappingID*
: Pointer to a **long** that receives the time slot enhancement mapping identifier. This value is only set if the episode is enhanced.

*lEpiEnhMappingID*
: Pointer to a **long** that receives the episode enhancement mapping identifier. This value is only set if the episode is enhanced.

**Remarks**

You can pass in NULL for either or both of the enhancement identifiers. In this case, **HasEnhancement** will still return a value that indicates whether the episode is enhanced, however it does not get values for the identifiers. The following code is an example of this usage:

```
pEPGEpisode->HasEnhancement(&fHasEnh, NULL, NULL)
```

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

# IEPGEpisode::IsContinuous

[This is preliminary documentation and subject to change.]

The **IsContinuous** method returns a value that indicates whether the episode is continuous.

```
HRESULT IsContinuous(
  VARIANT_BOOL * pf
);
```

**Parameters**

*pf*
: Pointer to a variable that receives VARIANT_TRUE if the episode is continuous, and VARIANT_FALSE otherwise.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### Remarks

If an episode is continuous, it means there is a single on-going episode for the channel broadcasting that episode. For example, the Program Guide channel broadcasts one continuous episode.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

## IEPGEpisode::IsOnLater

[This is preliminary documentation and subject to change.]

The **IsOnLater** method returns a value indicating whether the episode is on later.

```
HRESULT IsOnLater(
  VARIANT_BOOL * pf
);
```

### Parameters

*pf*
Pointer to a variable that receives VARIANT_TRUE if the episode is on later, and VARIANT_FALSE otherwise.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGEpisode::IsOnSoon

[This is preliminary documentation and subject to change.]

The **IsOnSoon** method returns a value that indicates whether the episode will be broadcast in the next five minutes.

```
HRESULT IsOnSoon(
  VARIANT_BOOL * pf
);
```

### Parameters

*pf*
: Pointer to a variable that receives VARIANT_TRUE if the episode will be broadcast in the next five minutes, and VARIANT_FALSE otherwise.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGEpisode::IsRecordItem

[This is preliminary documentation and subject to change.]

The **IsRecordItem** method returns a value indicating whether the episode is a record item.

```
HRESULT IsRecordItem(
  VARIANT_BOOL * pf
);
```

### Parameters

*pf*

Pointer to a variable that receives VARIANT_TRUE if a record reminder is set for the episode, and VARIANT_FALSE otherwise.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

**Remarks**

TV Viewer sets record items. Some record items, typically episodes that are for display purposes only, do not have a valid time slot associated with them — in other words, their TS Time Slot ID field equals zero. If the time slot identifier is zero, but **IsRecordItem** returns S_OK, this indicates that a record reminder is set but that there are no shows matching this record reminder in the Program Guide.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGEpisode::IsRemindItem

[This is preliminary documentation and subject to change.]

The **IsRemindItem** method returns a value indicating whether the episode is a remind item.

```
HRESULT IsRemindItem(
  VARIANT_BOOL * pf
);
```

**Parameters**

*pf*
Pointer to a variable that receives VARIANT_TRUE if a reminder is set for the episode, and VARIANT_FALSE otherwise.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### Remarks

Remind items are set by TV Viewer. Some remind items, typically episodes that are for display purposes only, do not have a valid time slot associated with them — in other words, their TS Time Slot ID fields equal zero. If the time slot identifier is zero, but **IsRemindItem** returns S_OK, this indicates that a reminder is set but that there are no shows matching this reminder in the Program Guide.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

## IEPGEpisode::Layout

[This is preliminary documentation and subject to change.]

The **Layout** method returns additional information about the episode's enhancement.

```
HRESULT Layout (
  long lEnhMappingID,
  BSTR * EnhTitle,
  BSTR * EnhLayout,
  BSTR * EnhAddress
);
```

### Parameters

*lEnhMappingID*
: **Long** that contains the enhancement mapping identifier. This corresponds to either the *lTSEnhMappingID* or *lEpiEnhMappingID* value returned by the **IEPGEpisode::HasEnhancement** method.

*EnhTitle*
: Pointer to a string that receives the title of the enhanced show.

*EnhLayout*
: Pointer to a string that receives the starting or default HTML layout page for the enhancement.

*EnhAddress*
: Pointer to a string that recieves the enhancement IP address. Triggers for the enhancement will be sent to this address.

### Remarks

This method is only valid for episodes that have enhancements. To test whether an episode is enhanced, call **IEPGEpisode::HasEnhancement**.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

## IEPGEpisode::PayPerView

[This is preliminary documentation and subject to change.]

The **PayPerView** method returns a value indicating whether the episode is a pay-per-view episode. This information corresponds to the TS Pay Per View field in the Guide database.

```
HRESULT PayPerView (
  VARIANT_BOOL * pf
);
```

### Parameters

*pf*
: Pointer to a variable that receives VARIANT_TRUE if the episode is a pay-per-view episode, and VARIANT_FALSE otherwise.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

## IEPGEpisode::RatingID

[This is preliminary documentation and subject to change.]

The **RatingID** method gets the rating identifier associated with the episode. This information corresponds to the E Rating ID field in the Guide database.

```
HRESULT RatingID(
  long* lRatingID
);
```

### Parameters

*lRatingID*
: Pointer to a **long** that receives the rating identifier.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

## IEPGEpisode::RemindRecordIdx

[This is preliminary documentation and subject to change.]

The **RemindRecordldx** method returns an index into the cache of show reminders in TV Viewer. This method is only valid for episodes for which the **IsRemindItem** or **IsRecordItem** method returns VARIANT_TRUE.

```
HRESULT RemindRecordIdx(
  long* idxRR
);
```

### Parameters

*idxRR*
: Pointer to a **long** that receives the index value. If the episode is not a record or a remind item, this value is –1.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGEpisode::Repetition

[This is preliminary documentation and subject to change.]

The **Repetition** method returns a value indicating the frequency, or type, of the reminder. This method is only valid for episodes for which the **IsRemindItem** or **IsRecordItem** method returns VARIANT_TRUE.

```
HRESULT Repetition(
  int* iRep
);
```

### Parameters

*iRep*

Pointer to an **int** that receives the repetition type flag. This flag can be one of the following named values.

| Value | Meaning |
|---|---|
| REMIND_DAILY | The reminder repeats each day. |
| REMIND_NONE | The reminder doesn't occur. This value may indicate that the episode does not have a show or record reminder set. |
| REMIND_ONCE | The reminder occurs once. |
| REMIND_WEEKDAYS | The reminder repeats each weekday. |
| REMIND_WEEKLY | The reminder repeats every week. |

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK.

Otherwise it returns an error code. For specific error code values see Winerror.h.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGEpisode::ThemeID

[This is preliminary documentation and subject to change.]

The **ThemeID** method retrieves the theme identifier of the episode. This information corresponds to the T Theme ID field in the Guide database.

```
HRESULT ThemeID(
  long* lThemeID
);
```

**Parameters**

*lThemeID*
Pointer to a **long** that receives the theme identifier.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# IEPGEpisode::TimeSlotID

[This is preliminary documentation and subject to change.]

The **TimeSlotID** method retrieves the time slot identifier of the episode. This information corresponds to the TS Time Slot ID field in the Guide database.

```
HRESULT TimeSlotID(
  long* lTimeSlotID
);
```

### Parameters

*lTimeSlotID*
Pointer to a **long** that receives the time slot identifier.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Import Library:** user-defined.

# Television System Services

[This is preliminary documentation and subject to change.]

The object library for Television System Services (TSS), Tssadmin.dll, provides television-related utility methods for Broadcast Architecture, as well as a television-related collection. Using the TSS object library, you can:

- Retrieve information about the Broadcast Architecture installation.
- Create and parse *show references*.
- Generate queries from a show reference that returns a list of all episodes that match the show reference.
- Load *enhancement* information into the *Guide database*.
- Set and delete *show reminders*.

For more information on using this library, see About Television System Services and Using Television System Services.

TSS contains methods that are not supported and are reserved for future use. For more information, see Reserved Methods in Television System Services.

## Reserved Methods in Television System Services

[This is preliminary documentation and subject to change.]

The object library for Television System Services (TSS), Tssadmin.dll, contains the following methods that are not supported and are reserved for future use:

- **ITelevisionServices::CreateUser**
- **ITelevisionServices::DeleteUser**
- **ITelevisionServices::RestrictionQueryName**

## About Television System Services

[This is preliminary documentation and subject to change.]

The TSS object library, Tssadmin.dll, provides a variety of television-related utility methods, as well as a television-related collection. The following topics group these items by related functionality:

- About the Property Methods describes methods used to return information about the Broadcast Architecture system.
- About the Enhancement-Loading Methods describes methods that load enhancement information into the Guide database.
- About the Show Reference Methods describes methods to create and parse show references.
- About the Scheduled Items Collection describes the collection of *show reminders*.

For related reference information on TSS, see the sections on the **ITelevisionServices** and **IScheduledItems** interfaces.

## About the Property Methods

[This is preliminary documentation and subject to change.]

The methods described in this topic retrieve information about the properties of the current TSS installation. The property methods are as follows:

- **ITelevisionServices::get_SystemFile**
- **ITelevisionServices::get_DatabaseFile**
- **ITelevisionServices::get_ScheduledItems**
- **ITelevisionServices::get_ClipboardFormat**
- **ITelevisionServices::get_ClipboardFormatName**

Your application can use these methods to locate database and system files on the user's computer, retrieve the show reference clipboard format and access the collection of currently-scheduled show reminders.

For example, Broadcast Architecture stores security information in Microsoft® Access database and workgroup files. You can get the path and filename of these files using the **ITelevisionServices::get_SystemFile** and **ITelevisionServices::get_DatabaseFile** methods.

Broadcast Architecture defines show reminders, a special type of task-scheduler task that reminder the user when a specific show is about to begin. You can retrieve a collection of all currently scheduled show reminders by calling **ITelevisionServices::get_ScheduledItems** method. For more information about this collection, see About the Scheduled Items Collection.

When Broadcast Architecture is installed, it saves information about a new clipboard format for show references in the registry. If your application knows the show reference format it can use the clipboard to cut and paste show references. The TSS object library provides two methods, **ITelevisionServices::get_ClipboardFormat** and **ITelevisionServices::get_ClipboardFormatName**, that you can use to retrieve the identifier and name of the show reference clipboard format.

For more information, see the following topics:

- Getting Information About TSS
- Getting Scheduled Reminders
- Using **IScheduledItems** to Schedule a Show Reminder

## About the Enhancement-Loading Methods

[This is preliminary documentation and subject to change.]

The TSS object library, Tssadmin.dll, provides methods that you can use to load *enhancement* information into the Guide database. The enhancement methods are as follows:

- **ITelevisionServices::LoadEnhancement**
- **ITelevisionServices::LoadEnhancementsFromFile**
- **ITelevisionServices::DeleteEnhancementFromID**
- **ITelevisionServices::DeleteOldEnhancements**
- **ITelevisionServices::RemapEnhancements**

An enhancement is additional content, typically HTML, that can be displayed to "enhance" or extend a show. When an application loads enhancement HTML files onto the user's machine, information about the enhancement, such as the location of the HTML files and which file to display first, should be loaded into the Guide database. This enables TV Viewer to properly display the enhancement.

There are two types of enhancements: episode and channel. An episode enhancement applies to a particular *episode* or set of episodes. A channel enhancement applies to all the shows on a particular *channel*. For example, a channel dedicated to science fiction might have an enhancement that enables the user to browse through a database of science fiction trivia. If a show has both an episode and a channel enhancement, TV Viewer displays the episode enhancement.

When TV Viewer tunes to a new channel, it queries the Guide database to see whether any of the enhancements listed in the database match the episode currently showing. TV Viewer displays the episode using the first matching enhancement file listed. If no episode enhancements exist for the show, TV Viewer searches for a channel enhancement listing. If neither an episode nor a channel enhancement apply to the current show, TV Viewer displays the broadcast using the default full-screen video layout.

If multiple enhancements exist for the current show, users can select which enhancement to view from the TV banner. The TV banner is a toolbar that displays across the top of the screen when the user clicks F10. It displays the channel number, episode title, show time, and informational icons. When the user clicks on the enhancement icon in the TV banner, a list of the available enhancements appears. From the list, users can select which enhancement TV Viewer displays with the show.

The HTML files that define an enhancement's layout can be loaded to the user's computer in two ways:

- Through digital data transmission means such as *Internet channel broadcasting* or encoding in the *vertical blanking interval* (VBI)
- As part of loading current *Program Guide* information into the Guide database

Typically, each set of enhancement layout files is stored in a separate directory. The default top-level directory for enhancement storage is C:\Program Files\TV Viewer\Layouts\, and layouts are typically stored in its subdirectories. For example, an episode of Star Trek might store its enhancement files in C:\Program Files\TV Viewer\Layouts\Trek123\.

Once the enhancement layout files reside on the user's computer, information about the enhancement files, such as their location and filename, must be stored in the Guide database. This enables TV Viewer to query the Guide database and locate the enhancement files associated with a particular show.

If the enhancement files were loaded onto the user's machine during an update of the Guide database, the Loadstub component of Broadcast Architecture loads information about the enhancement files into the Guide database. It does this by calling the **ITelevisionServices::LoadEnhancementsFromFile** method. For more information about Loadstub, see Updating the Guide Database.

If the enhancement files were loaded onto the user's machine through VBI encoding, the content provider sends an announcement trigger through the broadcast stream to the Enhancement Filter to indicate that a show is enhanced. (For more on announcements, see Announcements Overview.) When the Enhancement Filter receives information about a show's enhancement, it calls the **ITelevisionServices::LoadEnhancement** method to load this information into the Guide database.

If your application loads enhancement files onto the user's computer, it should also load information about these files into the Guide database by calling either **LoadEnhancement** or **LoadEnhancementsFromFile** methods.

TSS also provides the method **ITelevisionServices::DeleteEnhancementFromID**, which enables applications such as Announcement Listener to delete enhancement data from the Guide database. Typically, you do not need to call this method. Instead enhancement information is automatically deleted from the Guide database after it expires. Obsolete enhancement data is also periodically removed from the Guide database by Loadstub. To enable such cleanup, you must set an expiration date for any enhancement data that you load. You specify the enhancement data's expiration date in the *dateExpire* parameter of **LoadEnhancement** or in the Expiration Date column of the text file read by **LoadEnhancementsFromFile**.

You can delete expired enhancement data from the Guide database by calling **ITelevisionServices::DeleteOldEnhancements**.

The **ITelevisionServices::RemapEnhancements** method maps enhancements to their corresponding channels and episodes.

For more information about using the enhancement methods, see Loading Enhancement Information.

For more information about enhancements, see Video Enhancements.

## About the Show Reference Methods

[This is preliminary documentation and subject to change.]

Broadcast Architecture defines a data format, the *show reference*, to describe a broadcast episode or episodes using such information as the time and channel of the show and so on. A show reference is an ASCII string that contains multiple fields. Each field is separated from the next by an exclamation mark ( **!** ).

The TSS object library provides methods that build a show reference from its constituent fields, parse show references, and build database queries from a show reference. These queries can then be run against the Guide database to return the collection of episodes to which the show reference refers.

The show reference methods are as follows:

- **ITelevisionServices::MakeLocalBroadcastSchedule**
- **ITelevisionServices::MakeRemoteBroadcastSchedule**
- **ITelevisionServices::SplitBroadcastSchedule**
- **ITelevisionServices::SplitSimpleBroadcastSchedule**
- **ITelevisionServices::ResolveBroadcast**
- **ITelevisionServices::ResolveBroadcastInclusively**
- **ITelevisionServices::ResolveScheduledReminders**
- **ITelevisionServices::TuningSpaceNameFromNumber**
- **ITelevisionServices::TuningSpaceNumberFromName**

The **ITelevisionServices::MakeLocalBroadcastSchedule** method builds a show reference where a show's channel number, *tuning space*, and station are known. The **ITelevisionServices::MakeRemoteBroadcastSchedule** method also builds a show reference but substitutes wildcard values for the channel number, tuning space, and station.

You can use **MakeLocalBroadcastSchedule** if you are certain that all users will be located in your local broadcast area. Otherwise, you can use **MakeRemoteBroadcastSchedule**, which substitutes wildcard characters for the local broadcasting information, such as station and channel number. TV Viewer queries the user's Guide database to fill in the wildcard values with local broadcasting information when it uses a show reference created by **MakeRemoteBroadcastSchedule**.

A show reference is composed of several fields that contain information about the episode. Your application can access this information, for example the episode title, by parsing the show reference. To parse show references, the TSS object library offers the **ITelevisionServices::SplitBroadcastSchedule** method, which splits a show reference into its constituent fields. **SplitBroadcastSchedule** returns an **ITaskTrigger** interface pointer that specifies the date and task trigger information, such as run at 7:00 PM on day 23 of every month, starting

7/22/97. The **ITelevisionServices::SplitSimpleBroadcastSchedule** method performs the same service but returns the date information as a **DATE** variable. Although **SplitSimpleBroadcastSchedule** provides less information than **SplitBroadcastSchedule**, **SplitSimpleBroadcastSchedule** can be called by programs in the Microsoft® Visual Basic® programming system that cannot access **ITaskTrigger**.

The **ITelevisionServices::ResolveBroadcast** method generates a query from a show reference. By running this query on the Guide database, you can retrieve the episode or the list of episodes that match a particular show reference. This functionality is useful in situations where a reference refers to several episodes, which occurs if some of the reference's fields are wildcard values. For example, a show reference that specifies the time and title, but not the date, of a show can refer to all the weekly or monthly episodes of that show.

The **ITelevisionServices::ResolveBroadcastInclusively** method is identical in function to **ResolveBroadcast** except that it returns all matching episodes playing during the specified time in the show reference, instead of starting at that time.

The **ITelevisionServices::ResolveScheduledReminders** method returns a Data Access Objects (DAO) **QueryDef** query definition that generates a list of the show reminders scheduled. To locate more information on DAO, see Further General Information.

In addition, the TSS object library provides two methods, **ITelevisionServices::TuningSpaceNameFromNumber** and **ITelevisionServices::TuningSpaceNumberFromName**, which map the number of a tuning space in the Guide database to its human-readable name, such as cable, TV tuner, or satellite, and vice versa. You could use these methods to convert the tuning space identifiers into the device names to display a list to the user. When the user has selected a device name, you could call **TuningSpaceNumberFromName** to recover the tuning space in order to reference the tuning space in the Guide database or to tune to a channel in that tuning space.

To locate more information about **ITaskTrigger**, see Further Information on Television Services for the Client.

For more information, see the following topics:

- Creating a Show Reference
- Parsing a Show Reference
- Resolving a Show Reference
- Mapping a Tuning Space Name to an Identifier

## About the Scheduled Items Collection

[This is preliminary documentation and subject to change.]

A *show reminder* is a task scheduler task that runs just before a show starts and displays a dialog box to the user, informing the user the show is about to begin. The currently set reminders are enumerated by the Scheduled Items collection returned by the **ITelevisionServices::get_ScheduledItems** method.

Once you have a reference to the collection, you can use the methods of the **IScheduledItems** interface to set, delete, or enumerate the scheduled shows. For more information, see Using **IScheduledItems** to Schedule a Show Reminder.

However, show reminders set using **IScheduledItems** do not appear in the TV Viewer user interface. In other words, users cannot search for or set reminders using the TV Viewer search page. To create scheduled reminders that are visible from TV Viewer, you must use the **ITVViewer::SetReminder** method. For more information, see Using **ITVViewer** to Schedule a Show Reminder.

# Using Television System Services

[This is preliminary documentation and subject to change.]

The TSS object library, Tssadmin.dll, provides a variety of utility methods for Broadcast Architecture–aware applications. The following topics describe how to use the different areas of functionality provided by these methods:

- Getting Information About TSS
- Loading Enhancement Information
- Creating a Show Reference
- Parsing a Show Reference
- Resolving a Show Reference
- Mapping a Tuning Space Name to an Identifier
- Getting Scheduled Reminders
- Using IScheduledItems to Schedule a Show Reminder

## Getting Information About TSS

[This is preliminary documentation and subject to change.]

The TSS object library provides methods that you can use to acquire information about the current Broadcast Architecture installation.

To get the path and name of the system workgroup information file that is used for TSS database security, your application calls the **ITelevisionServices::get_SystemFile** method. With this information, you can set the system database of the Jet database engine. The path and file name of the Guide database can be retrieved by calling the **ITelevisionServices::get_DatabaseFile** method. With this information, you can open the database. To locate more information on working with Jet databases, see Further Information on Television Services for the Client.

Broadcast Architecture saves information about a new clipboard format for show references in the registry. The TSS object library provides methods to retrieve information about this format. With this information, your applications can cut and paste show references using the clipboard. To retrieve the clipboard format identifier, your application calls the **ITelevisionServices::get_ClipboardFormat** method. To retrieve the clipboard format name, your application calls **ITelevisionServices::get_ClipboardFormatName**.

For more information, see About the Property Methods.

## Loading Enhancement Information

[This is preliminary documentation and subject to change.]

When TV Viewer displays an enhanced show, it looks up information about the enhancements, such as which layout files to use and in which directory they are stored, in the Guide database. In order for the enhanced show to display properly, this information about the enhancement files must be loaded into the Guide database before the show is broadcast.

The **ITelevisionServices** interface provides two methods that you can use to load enhancement information into the Guide database, **ITelevisionServices::LoadEnhancement** and **ITelevisionServices::LoadEnhancementsFromFile**.

**LoadEnhancement** loads information about a single enhancement into the database, as specified by parameters passed in the method call. For example, if you were writing an application that downloaded a set of enhancement files for a show onto the user's computer, you could then call **LoadEnhancement** to store information about those files in the Guide database so that TV Viewer would be able to locate and properly display the enhancements.

If you are loading information about multiple enhancements you can use the **LoadEnhancementsFromFile** method. It loads information into the database about multiple enhancements, as listed in the text file the method call points to. Because **LoadEnhancementsFromFile** uses the JET text Installable Sequential Access Module (ISAM) to parse the text file, you must include a Schema.ini file in the same directory as the text file that contains enhancement information. For examples of valid Schema.ini files, see the Remarks section of the **LoadEnhancementsFromFile** reference entry.

Both **LoadEnhancement** and **LoadEnhancementsFromFile** programmatically determine whether an enhancement applies to a channel or to a specific episode. If the show reference parameter contains an episode title, the method sets the indicated enhancement to apply to the specified episode or episodes.

If the show reference parameter does not contain an episode title, the enhancement is assumed to be a channel enhancement. In this case, **LoadEnhancement** or **LoadEnhancementsFromFile** searches the *tuning space* for the union of station, network, and channel number specified in *bstrShowReference* or *Show Reference*. The method then sets the enhancement to apply to all channel records that match this union.

A third function, **ITelevisionServices::DeleteEnhancementFromID**, enables you to remove enhancement information from the Guide database. You could, for example write a trigger event handler that would respond to a delete-enhancement event sent by the broadcast provider and delete the specified enhancement information from the database. However, this method is not often used. Typically enhancements are deleted automatically when they expire by internal components of the Broadcast Architecture system.

You can delete expired enhancement data from the Guide database by calling **ITelevisionServices::DeleteOldEnhancements**.

For more information, see About the Enhancement-Loading Methods.

## Creating a Show Reference

[This is preliminary documentation and subject to change.]

To create a show reference from episode data, you can use either the **ITelevisionServices::MakeLocalBroadcastSchedule** or **ITelevisionServices::MakeRemoteBroadcastSchedule** method. With the first method, you must pass in local broadcast information, including the channel number, local station, and tuning space. With the second, you can create a show reference that substitutes wildcard values for these entries. Substituting wildcard values causes applications with functionality like that of TV Viewer to query the Guide database to match an episode or episodes using only the episode title, network, and show time.

**MakeRemoteBroadcastSchedule** is useful when you do not know which broadcast area might be local. For example, you can access a Web application from any broadcast area. If your Web application does not know or calculate in the user's broadcast area, it should call **MakeRemoteBroadcastSchedule**.

Note that you can achieve the same effect by calling **MakeLocalBroadcastSchedule** and passing in wildcard values for the *Channel*, *Station*, and *TuningSpace* parameters.

For more information, see the following topics:

- Show Reference Format
- Resolving a Show Reference
- About the Show Reference Methods

## Parsing a Show Reference

[This is preliminary documentation and subject to change.]

A show reference is composed of several fields that contain information about the episode or episodes. Your application can access the episode information, for example the episode title, by parsing the show reference. To do this, you can use either the **ITelevisionServices::SplitBroadcastSchedule** or **ITelevisionServices::SplitSimpleBroadcastSchedule** method. Both methods split a show reference into its constituent fields. However, the first method returns the trigger information, date information, or both as a pointer to a **ITaskTrigger** interface, and the second returns only the date information in a **DATE** variable.

If you are using the Microsoft® Visual Basic® programming system to write your application, you should use **SplitSimpleBroadcastSchedule** instead of **SplitBroadcastSchedule**. Although **SplitSimpleBroadcastSchedule** returns less information about the show reference, it is preferred for Visual Basic applications, which cannot access **ITaskTrigger**.

For more information on show reference fields, see Show Reference Format. To locate more information about **ITaskTrigger**, see Further Information on Television Services for the Client.

For more information, see About the Show Reference Methods.

## Resolving a Show Reference

[This is preliminary documentation and subject to change.]

You can use the **ITelevisionServices::ResolveBroadcast** method to generate a Guide database query that lists all the episodes that match a show reference. Doing so is useful, for example, if your application needs to resolve a reference created with **MakeRemoteBroadcastSchedule** to the correct local broadcast information.

To generate a query that lists all the matching episodes that have started before and ended after a particular time, you can call the **ITelevisionServices::ResolveBroadcastInclusively** method.

For more information, see Creating a Show Reference and About the Show Reference Methods.

## Mapping a Tuning Space Name to an Identifier

[This is preliminary documentation and subject to change.]

A *tuning space* is a Broadcast Architecture feature that enables a *broadcast client* to resolve overlapping channel numbers from multiple broadcast sources. For example, suppose a computer has both an analog television tuner and a satellite tuner card installed. If the two broadcast systems each have a channel 1, conflicts occur. To prevent such conflicts, Broadcast Architecture defines each input source as a separate tuning space, assigning it a unique identifier.

Each tuning space has both a unique numerical identifier, and a human-readable name such as analog tuner, cable, or satellite.

You can use methods of the TSS object library to map a tuning space identifier to its user-readable name and vice versa. For example, if your application needed to present a list of tuning spaces to the user, you could look up the tuning spaces in the Guide database, and then map each tuning space to a human-readable name. Conversely, when the user selected a tuning space, your application would need to map the tuning space name back to its identifier before it could tune to a channel in that tuning space.

To map an identifier to the tuning space name, your application calls **ITelevisionServices::TuningSpaceNameFromNumber**. To map a name to its corresponding identifier, your application calls **ITelevisionServices::TuningSpaceNumberFromName**.

For more information, see Overlapping Channels and About the Show Reference Methods.

## Getting Scheduled Reminders

[This is preliminary documentation and subject to change.]

You can use the TSS object library to retrieve a collection or enumeration of all currently scheduled show reminders, or a list of the reminders set. You can then set, edit, and delete the scheduled reminders.

To access all scheduled reminders, your application calls the **ITelevisionServices::get_ScheduledItems** method. Doing so populates and returns a collection of **ScheduledItems** objects, each one corresponding to a scheduled reminder. Your application can then set, edit, and delete reminders using the methods of the **IScheduledItems** interface.

To retrieve a list of scheduled reminders set, your application calls the **ITelevisionServices::ResolveScheduledReminders** method. Doing so generates a DAO **QueryDef** query, which you can then run on the Guide database to generate the list of reminders set.

For more information, see About the Scheduled Items Collection.

## Using IScheduledItems to Schedule a Show Reminder

[This is preliminary documentation and subject to change.]

As described in Scheduling Show Reminders, Broadcast Architecture provides two ways to programmatically schedule show reminders. The first process is described in Using ITVViewer to Schedule a Show Reminder. You can also create a show reminder by using the **IScheduledItems** interface.

Using **IScheduledItems** is better for applications that do not connect to TV Viewer. For example, an enhancement that is a Microsoft® ActiveX™ control or Java applet on a World Wide Web page can schedule a reminder for the show related to the enhancement by using **IScheduledItems**. **IScheduledItems** is much simpler to start, load, and release than the **ITVViewer** interface. For more information on connecting to TV Viewer, see Connecting to TV Viewer.

However, reminders set using **IScheduledItems** are not always visible in the the TV Viewer user interface. In other words, they do not automatically appear in the **MyReminders** list of the search page in TV Viewer. In order for a TSS-set reminder to be visible in TV Viewer it must meet certain standards. These standards are specified in Setting a Reminder that Appears in TV Viewer.

If a reminder set using **IScheduledItems** does not appear in the TV Viewer reminders list, users will be unable to delete the reminder using TV Viewer. In this case, you should set an expiration date for such a reminder by creating a **TASK_TRIGGER** structure and specifying values for the **wEndYear**, **wEndMonth**, and **wEndDay** members. Alternatively, you can provide an application that the user can use to delete the reminder. To locate more information about **TASK_TRIGGER**, see Further Information on Television Services for the Client.

Although you cannot view a reminder set with **Add** or **AddFromQuery** from TV Viewer, you can still use TV Viewer to display that reminder to the user when it runs. To do so, simply specify TV Viewer (Tvx.exe) as the reminder application when you set the show reminder.

Before you set a reminder, you must gather information about the show reminder you wish to set. This information is passed into the parameters of the **Add** or **AddFromQuery** method. The method formats this information into the show reminder format and then schedules the reminder with the Task Scheduler.

**To gather information for show reminder parameters**

1. Create a show reference by passing the episode information to either the **ITelevisionServices::MakeLocalBroadcastSchedule** or **ITelevisionServices::MakeRemoteBroadcastSchedule** method.
2. If you know the duration of the episode in minutes, add this information to the *TVViewerParameters* string, later to be passed to the **Add** method in the *Parameters* parameter. If you do not know the episode duration, specify *TVViewerParameters* as

   "*Flag*!*Type*!"

   where
   *Flag*
   : Indicates whether to display this reminder in the TV Viewer reminders list. If the flag is set to **tvviewer**, and the reminder meets the standards specified in Setting a Reminder that Appears in TV Viewer, TV Viewer displays the reminder in its list. Otherwise it does not.

*Type*

Indicates the type of reminder set, either Remind or Record.

If the reminder is a record reminder, you should use the Task Scheduler to set the TASK_FLAG_SYSTEM_REQUIRED flag for the reminder. This causes TV Viewer to tune to the channel even if the system is sleeping. Otherwise, if the system is sleeping, TV Viewer will not wake up to run the record reminder.

In addition, if the record reminder has an application associated with it that automates tuning the VCR this application should be specified in the StartRecordingApp and/or EndRecordingApp values under this registry key:

**HKLM\Software\Microsoft\TV Services\Explorer\**

The TASK_FLAG_SYSTEM_REQUIRED flag should not be set for standard show reminders. Version 1.0 of Broadcast Architecture does not handle show reminders that go off while the system is sleeping.

For more information see Setting a Show Reminder and Setting a Record Reminder.

**Example:** "tvviewer!Remind!" or "tvviewer!Record!"

3. If the application to display the reminder is TV Viewer, specify the path for Tvx.exe as *path* in the string *path***Tvx.exe**. Your application later passes this information to **Add** in the *Directory* parameter. You can retrieve this information by calling the **ITelevisionServices::get_DatabaseFile** method to locate the Guide database, installed in the same directory as Tvx.exe.

   If the reminder application is not TV Viewer, specify the application's working directory and name as the string *pathfilename*, where *path* is the path and *filename* the file name.

**Note** Record reminders using TV Viewer should not specify Tvx.exe as their display application. Instead, they should use the application listed in this registry value:

**HKLM\Software\Microsoft\TV Services\Explorer\HelperApp**

If that value is not defined, they should use Tvwakeup.exe.

4. Determine the number of minutes before the scheduled show that the reminder should be displayed, later to be passed to **Add** in the *AdvanceMinutes* parameter.
5. Schedule the show reminder by calling **Add**.

When the reminder runs, the application that you specified in the show reminder parameters, usually TV Viewer, displays to the user a notification that the show is on. The details of the show reminder that you specified when setting the reminder are passed to the reminder application as command-line arguments when it runs the reminder. For more information about show reminder details passed to the reminder application, see Show Reminder Format.

The application should use the **ITelevisionServices::ResolveBroadcast** method to resolve the passed-in show reference to a television episode.

For more information, see About the Scheduled Items Collection and Scheduling Show Reminders.

# Television System Services Reference

[This is preliminary documentation and subject to change.]

The primary Automation class in Television System Services (TSS), Tssadmin.dll, is **ITelevisionServices**, derived from the **IDispatch** interface. For error handling, **ITelevisionServices** supports the **IErrorInfo** Automation interface, described in the Automation component in the Platform Software Development Kit (SDK).

TSS also provides the **IScheduledItems** interface, which contains methods that you can use to set, edit, or delete show reminders.

## ITelevisionServices

[This is preliminary documentation and subject to change.]

**ITelevisionServices** provides the following methods to enable your application to retrieve read-only properties associated with this interface.

### When to Implement

**ITelevisionServices** is implemented in Tssadmin.dll.

### When to Use

Call the methods of **ITelevisionServices** to access television service-related utilities.

### Methods in Vtable Order

| **IUnknown Methods** | **Description** |
|---|---|
| **QueryInterface** | Returns pointers to supported interfaces |
| **AddRef** | Increments reference count |
| **Release** | Decrements reference count |

| **ITelevisionServices** | **Description** |
|---|---|
| get_SystemFile | This read-only property contains the name of the Microsoft® Jet database .mdw file that provides the security model for TSS. Applications that use TSS must open the database file with this system file, not the default system file recorded in the registry. Applications do so by setting the **SystemDB** property of the Data Access Object (DAO) database engine before opening the database. |
| get_DatabaseFile | This read-only property contains the name of the Jet .mdb file that contains the Program Guide data. |
| get_ScheduledItems | This read-only property returns the **IScheduledItems** collection. These are all the reminders currently active in the system. |
| ResolveBroadcast | Creates a DAO **QueryDef** that represents the specified show reference, for a show that starts at the specified start time |
| SplitBroadcastSchedule | Parses a show reference into its constituent parts |
| SplitSimpleBroadcastSchedule | Performs the same function as the **SplitBroadcastSchedule** method, except that the time returned is a single date |
| MakeLocalBroadcastSchedule | Returns a correctly formatted show reference |
| MakeRemoteBroadcastSchedule | Provides a wrapper for the **MakeLocalBroadcastSchedule** method that substitutes wildcard values for computer- or locality-specific members |
| TuningSpaceNameFromNumber | Maps the numeric representation of a video source in the Guide database to a user-readable string (such "Viacom") |
| TuningSpaceNumberFromName | Maps a user-readable string denoting a tuning space (such as "Viacom") to the index by which that tuning space is known in the Guide database |

| | |
|---|---|
| **ResolveScheduledReminders** | Creates a DAO **QueryDef** that represents the television shows that match the scheduled reminders |
| **LoadEnhancement** | Loads data about an enhancement into the Guide database. |
| **LoadEnhancementsFromFile** | Loads data about multiple enhancements into the Guide database. |
| **DeleteEnhancementFromID** | Deletes data about an enhancement from the Guide database. |
| **DeleteOldEnhancements** | Deletes expired enhancement information from the Guide database. |
| **RemapEnhancements** | Remaps enhancement information in the Guide database. |
| **get_ClipboardFormat** | Returns the identifier for the Clipboard format for show references |
| **get_ClipboardFormatName** | Returns the registered name of the Clipboard format for show references |
| **ResolveBroadcastInclusively** | Creates a DAO **QueryDef** that represents the specified show reference, which contains the specified start time |

**Remarks**

**ITelevisionServices** is derived from the **IDispatch** interface. To locate more information on **IDispatch**, a Component Object Model (COM) interface, see Further Information on Television Services for the Client.

**See Also**

**IScheduledItems**

# ITelevisionServices::get_ClipboardFormat

[This is preliminary documentation and subject to change.]

The **get_ClipboardFormat** method returns the registration identifier of the Clipboard format for show references. The registration identifier is the value returned by the **RegisterClipboardFormat** function, part of the Microsoft® Win32® application programming interface (API), when the show references format is registered.

```
HRESULT get_ClipboardFormat(
  long * plRetVal  // out
);
```

### Parameters

*plRetVal*
: Pointer to a **long** value that identifies the registered show references Clipboard format.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

# ITelevisionServices::get_ClipboardFormatName

[This is preliminary documentation and subject to change.]

The **get_ClipboardFormatName** method returns the name under which the show references Clipboard format has been registered using the Win32 **RegisterClipboardFormat** function.

```
HRESULT get_ClipboardFormatName(
  BSTR  * pbstrName      // out
);
```

### Parameters

*pbstrName*
: Pointer to a string containing the name under which the show references Clipboard format is registered.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

## ITelevisionServices::get_DatabaseFile

[This is preliminary documentation and subject to change.]

The **get_DatabaseFile** function retrieves a read-only value that specifies the name of the Jet .mdb file that contains the Program Guide data.

```
HRESULT get_DatabaseFile(
  BSTR *pbstrRetVal
);
```

### Parameters

*pbstrRetVal*
A pointer to a **BSTR** that receives the file name.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

# ITelevisionServices::get_ScheduledItems

[This is preliminary documentation and subject to change.]

The **get_ScheduledItems** function returns a read-only **IScheduledItems** collection that contains all the reminders currently active in the system.

```
HRESULT get_ScheduledItems(
  IScheduledItems **ppScheduledRet
);
```

**Parameters**

*ppScheduledRet*

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

# ITelevisionServices::get_SystemFile

[This is preliminary documentation and subject to change.]

The **get_SystemFile** function retrieves a read-only value that specifies the name of the Jet .mdw file used by TSS. Applications that use TSS must open the database file with this system file, not the default system file recorded in the registry. Applications do so by setting the **SystemDB** property of the DAO database engine before opening the database.

```
HRESULT get_SystemFile(
  BSTR *pbstrRetVal
);
```

**Parameters**

*pbstrRetVal*
A pointer to a **BSTR** that receives the file name.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

# ITelevisionServices::MakeLocalBroadcastSched

[This is preliminary documentation and subject to change.]

The **MakeLocalBroadcastSchedule** method returns a correctly formatted show reference. The **MakeLocalBroadcastSchedule** parameters contain all the information necessary for making such a reference.

```
HRESULT MakeLocalBroadcastSchedule(
  BSTR   EpisodeTitle,   // in
  short   Channel,       // in
  BSTR   Network,        // in
  BSTR   Station,        // in
  long   TuningSpace,   // in
  VARIANT  Time,         // in
  BSTR   *pbstrRetVal    // out
);
```

### Parameters

*EpisodeTitle*
Title for the show reference.
*Channel*
Channel for the show reference. To specify any channel, set this parameter to the wildcard value of –1.
*Network*
Network for the show reference.
*Station*

Station for the show reference. To specify any station, set this parameter to the wildcard value of NULL.

*TuningSpace*

Tuning space for the show reference. To specify any tuning space, set this parameter to the wildcard value of –1.

*Time*

Time for the show reference. The *Time* parameter is a **VARIANT** that can contain either a **DATE** value or an **ITaskTrigger** pointer. The former can be used from the Microsoft® Visual Basic® programming system; the latter permits repeating, or ambiguous schedules.

You can create a show reference with an ambiguous date or time by passing in a **ITaskTrigger** object where the unspecified date and/or time members of the **TASK_TRIGGER** structure are set to –1. For more information, see Ambiguous Triggers.

The **DATE** data type does not support ambiguous date or time values.

*pbstrRetVal*

Address where this method returns a properly formatted show reference.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### Remarks

The **MakeLocalBroadcastSchedule** method does not perform any validation of the existence of the show or shows specified or more than rudimentary validation of its arguments.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

### See Also

**MakeRemoteBroadcastSchedule**, **SplitBroadcastSchedule**, **SplitSimpleBroadcastSchedule**

## ITelevisionServices::MakeRemoteBroadcastSche

[This is preliminary documentation and subject to change.]

The **MakeRemoteBroadcastSchedule** method is a wrapper for the **MakeLocalBroadcastSchedule** method that substitutes wildcard values for computer- or locality-specific members.

```
HRESULT MakeRemoteBroadcastSchedule(
  BSTR   EpisodeTitle,  // in
  BSTR   Network,       // in
  VARIANT  Time,        // in
  BSTR   *pbstrRetVal  // out
);
```

## Parameters

*EpisodeTitle*
Title for the show reference.

*Network*
Network for the show reference.

*Time*
Time for the show reference. The *Time* parameter is a **VARIANT** that can contain either a **DATE** value or an **ITaskTrigger** pointer. The former can be used from Visual Basic; the latter permits repeating, or ambiguous schedules.

You can create a show reference with an ambiguous date or time by passing in a **ITaskTrigger** object where the unspecified date and/or time members of the **TASK_TRIGGER** structure are set to –1. For more information, see Ambiguous Triggers.

The **DATE** data type does not support ambiguous date or time values.

*pbstrRetVal*
Address where this method returns a properly formatted show reference.

## Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

## Remarks

The **MakeRemoteBroadcastSchedule** method does not perform any validation of the existence of the show or shows specified or more than rudimentary validation of its arguments.

## QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

### See Also

**MakeLocalBroadcastSchedule**, **SplitBroadcastSchedule**, **SplitSimpleBroadcastSchedule**

# ITelevisionServices::ResolveBroadcast

[This is preliminary documentation and subject to change.]

The **ResolveBroadcast** method creates a DAO **QueryDef** definition that represents the specified show reference.

```
HRESULT ResolveBroadcast(
  BSTR  Workspace,      // in
  BSTR  QueryName,      // in
  BSTR  ShowReference   // in
);
```

### Parameters

*Workspace*
: Name of the DAO workspace in which to create the query. This can be either a preexisting, or newly created workspace.

*QueryName*
: Name of the new query.

*ShowReference*
: Show reference used to generate the query.

If *ShowReference* is unambiguous, the query generated will represent a single episode.

In contrast, ambiguous show references can represent multiple or repeating broadcasts. For example, if you wanted to generate a query that returned each weekly episode of a show, you could specify the time and channel of that show, leaving the date ambiguous.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

### Remarks

If a **QueryDef** of the same name already exists, calling **ResolveBroadcast** deletes it. The method **ResolveBroadcast** can return an error. The **QueryDef** can have an empty result set if the *ShowReference* specified does not match anything in the database without causing an error. The caller of this method must delete the **QueryDef** object after use.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

## ITelevisionServices::ResolveBroadcastInclusivel

[This is preliminary documentation and subject to change.]

The **ResolveBroadcastInclusively** method creates a DAO **QueryDef** definition that represents the specified show reference. If a start time is specified in the show reference, this method returns all matching episodes started before or ending after that time. The **ResolveBroadcastInclusively** functionality differs from **ResolveBroadcast**, which only returns matching episodes started at the specified start time.

```
HRESULT ResolveBroadcastInclusively(
  BSTR  Workspace,     // in
  BSTR  QueryName,     // in
  BSTR  ShowReference  // in
);
```

**Parameters**

*Workspace*
: Name of the DAO workspace in which to create the query. This can be either a preexisting, or newly created workspace.

*QueryName*
: Name of the new query.

*ShowReference*
: Show reference used to generate the query. If *ShowReference* is unambiguous, the query generated represents a single episode. In contrast, an ambiguous show reference can represent multiple or repeating broadcasts. For example, to generate a query that returns each weekly episode of a show, specify the time and channel of that show leaving the date ambiguous.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

**Remarks**

This method is called internally by the enhancement filter.

As an example of **ResolveBroadcastInclusively** functionality, if the start time specified in the show reference is 11:15 A.M., **ResolveBroadcastInclusively** creates a **QueryDef** that returns shows that start any time before and end any time after 11:15 A.M. If a **QueryDef** of the same name already exists, calling **ResolveBroadcastInclusively** deletes it.

**ResolveBroadcastInclusively** can return an error. The **QueryDef** can have an empty result set if the *ShowReference* specified does not match anything in the database without causing an error. The caller of this method must delete the **QueryDef** object after use.

**ResolveBroadcastInclusively** ignores the show reference's end time value.

# ITelevisionServices::ResolveScheduledReminder

[This is preliminary documentation and subject to change.]

The **ResolveScheduledReminders** method creates a DAO **QueryDef** definition that represents the television shows that match the scheduled reminders.

```
HRESULT ResolveScheduledReminders(
  BSTR  Workspace,   // in
  BSTR  QueryName,   // in
  BSTR  Reserved     // in
);
```

### Parameters

*Workspace*
Name of the DAO workspace in which to create the query. This can be either a preexisting, or newly created workspace.

*QueryName*
Name of the new query.

*Reserved*
This parameter should be an empty string.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

### Remarks

The caller of the **ResolveScheduledReminders** method must delete the **QueryDef** object after use.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

# ITelevisionServices::SplitBroadcastSchedule

[This is preliminary documentation and subject to change.]

The **SplitBroadcastSchedule** method parses a show reference into its constituent parts.

```
HRESULT SplitBroadcastSchedule(
  BSTR   ShowReference,      // in
  BSTR * EpisodeTitle,       // out
  short *   Channel,         // out
  BSTR * Network,            // out
  BSTR * Station,            // out
  long *   TuningSpace,      // out
  IUnknown ** TaskTrigger    // out
);
```

**Parameters**

*ShowReference*
Show reference to parse.
*EpisodeTitle*
Pointer to a BSTR where this method returns the title for the show reference.
*Channel*
Pointer to a short where this method returns the channel for the show reference.
*Network*
Pointer to a BSTR where this method returns the network for the show reference.
*Station*
Pointer to a BSTR where this method returns the station for the show reference.
*TuningSpace*
Pointer to a long where this method returns the tuning space for the show reference.
*TaskTrigger*
Pointer to an interface pointer where this method returns the time information for the show reference.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### Remarks

The **SplitBroadcastSchedule** method returns an error if the syntax of the show reference string is faulty.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

### See Also

**MakeLocalBroadcastSchedule**, **MakeRemoteBroadcastSchedule**, **ITelevisionServices::SplitSimpleBroadcastSchedule**

## ITelevisionServices::SplitSimpleBroadcastSched

[This is preliminary documentation and subject to change.]

The **SplitSimpleBroadcastSchedule** method parses a show reference into its constituent parts. It performs the same function as the **SplitBroadcastSchedule** method, except that the time returned is a single date.

```
HRESULT SplitSimpleBroadcastSchedule(
  BSTR   ShowReference,   // in
  BSTR * EpisodeTitle,    // out
  short *  Channel,       // out
  BSTR * Network,         // out
  BSTR * Station,         // out
  long *  TuningSpace,    // out
  DATE * Time             // out
);
```

### Parameters

*ShowReference*
Show reference to parse.
*EpisodeTitle*

Pointer to a BSTR where this method returns the title for the show reference.

*Channel*

Pointer to a short where this method returns the channel for the show reference.

*Network*

Pointer to a BSTR where this method returns the network for the show reference.

*Station*

Pointer to a BSTR where this method returns the station for the show reference.

*TuningSpace*

Pointer to a long where this method returns the tuning space for the show reference.

*Time*

Pointer to a DATE where this method returns the time information for the show reference.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h.

### Remarks

The **SplitSimpleBroadcastSchedule** method is provided for Visual Basic programmers who cannot manipulate objects with the **ITaskTrigger** interface. This method returns an error if the syntax of the show reference string is faulty.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

### See Also

**MakeLocalBroadcastSchedule**, **MakeRemoteBroadcastSchedule**, **ITelevisionServices::SplitBroadcastSchedule**

## ITelevisionServices::TuningSpaceNameFromNu

[This is preliminary documentation and subject to change.]

The **TuningSpaceNameFromNumber** method maps the numeric representation of a video source in the Program Guide database to a user-readable string (for example, "Viacom").

```
HRESULT TuningSpaceNameFromNumber(
  long   TuningID,      // in
  BSTR * pbstrRetVal  // out
);
```

### Parameters

*TuningID*
  Numeric identifier of the tuning space.
*pbstrRetVal*
  Address where this method returns the tuning-space name string.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

### See Also

**TuningSpaceNumberFromName**

## ITelevisionServices::TuningSpaceNumberFromN

[This is preliminary documentation and subject to change.]

The **TuningSpaceNumberFromName** method maps a user-readable string denoting a tuning space (for example, "Viacom") to the index by which that tuning space is known in the Program Guide database.

```
HRESULT TuningSpaceNumberFromName(
  BSTR  Name,       // in
  long * plRetVal   // out
);
```

### Parameters

*Name*
  Name of the video source for the tuning space.

*plRetVal*
    Address where this method returns the tuning space index.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

**See Also**

**TuningSpaceNameFromNumber**

# ITelevisionServices::LoadEnhancement

[This is preliminary documentation and subject to change.]

The **LoadEnhancement** function loads a single enhancement into the Guide database.

```
HRESULT LoadEnhancement(
  Workspace *piDAOWorkspace,
  BSTR bstrEnhancementID,
  BSTR bstrShowReference,
  BSTR bstrTitle,
  BSTR bstrPreloadURL,
  BSTR bstrAddress,
  DATE dateExpire,
  DWORD fLoud
);
```

**Parameters**

*piDAOWorkspace*
    A pointer to a DAO workspace. This can be either a preexisting, or newly created workspace.
*bstrEnhancementID*
    Specifies a unique identifier for the enhancement.

    If you specify *bstrEnhancementID* as an empty string (""), the **LoadEnhancement** method automatically generates a unique identifier (GUID) for the enhancement. This option should only be used if the calling application has no need to further reference the enhancement entry.

*bstrShowReference*

The show reference string of the broadcast to which the enhancement applies. You can create a show reference by calling either the **ITelevisionServices::MakeLocalBroadcastSchedule** or **ITelevisionServices::MakeRemoteBroadcastSchedule** methods.

If *bstrShowReference* contains an episode title, the enhancement is treated as an episode enhancement, and is only set for the episode(s) specified.

If the show reference contains an empty episode title (""), it is assumed to be a channel enhancement, and the **ITelevisionServices::LoadEnhancement** and **ITelevisionServices::LoadEnhancementsFromFile** methods search the tuning space for the union of the station, network, and channel number parameters specified in *bstrShowReference*. The method then sets an enhancement for all matching channel records.

Example: "1997/7/29!26673/7175/9690!0:30!0!0!0!0!9690!9690!4096!7040!"!'CBUT'!3!54! Fresh Prince of Bel-Air".

For an exact description of show reference string syntax, see Show Reference Format.

*bstrTitle*

Specifies the title of the enhancement.

This value is displayed by TV Viewer when an episode has more than one enhancement available. When the user clicks the enhancement icon in the TV banner, TV Viewer displays a list of the enhancement titles, enabling the user to select which enhancement to display.

Example: "Fresh Prince Enhanced"

*bstrPreloadURL*

Specifies the URL of the HTML file that contains the Layout of the enhancement. If you do not specify a complete path, such as "C:\MyEnhance\", the URL is resolved relative to "C:\Program Files\TV Viewer\Layouts".

For example, if you specify *bstrPreloadURL* as "Cspan\Cspan.htm" it will be resolved to C:\Program Files\TV Viewer\Layouts\Cspan\Cspan.htm".

*bstrAddress*

A string that specifies the network card address, multicast address and port information used to connect to the enhancement stream. It must have the following format: "*netcard_address***\t***multicast_address***\t***multicast_port*".

Example: "125.125.125.125\t225.225.225.255\t10024"

*dateExpire*

The date on which the enhancement expires.

All enhancements should have an expiration date. This enables Loadstub to delete obsolete enhancement data from the Guide database.

*fLoud*

**DWORD** that indicates additional behavior for **LoadEnhancement**. This can be one or more of the following flags:

| **Value** | **Description** |
|---|---|
| LE_LOUD | If this flag is set, **LoadEnhancement** sends a Windows broadcast message to information applications that the data has changed.<br><br>Otherwise, **LoadEnhancement** does not send a message. |
| LE_DONTOVERWRITE | If this flag is set, and the enhancement data already exists in the database, **LoadEnhancement** does not re-load the data.<br><br>The value of *bstrEnhancementID* is used to determine whether the data is already loaded in the database. |

If this value is S_TRUE, TV Viewer is notified. If this value is S_FALSE, TV Viewer is not notified.

Note that even when *fLoud* is set to S_TRUE, TV Viewer is notified only if the loaded enhancmenent matches current program guide data. For example, if an enhancement for a December show is loaded in October, the enhancement is saved in the database, but TV Viewer is not notified. This is because the data is for the December episode is not yet listed in the Guide database.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

### See Also

**ITelevisionServices::LoadEnhancementsFromFile**,
**ITelevisionServices::DeleteEnhancementFromID**

# ITelevisionServices::LoadEnhancementsFromFi

[This is preliminary documentation and subject to change.]

The **LoadEnhancementsFromFile** method loads the enhancements listed in a text file into the Guide database.

```
HRESULT LoadEnhancementsFromFile(
  Workspace *piDAOWorkspace,
  BSTR bstrEnhancementsFile
);
```

### Parameters

*piDAOWorkspace*
A pointer to a DAO workspace. This can be either a preexisting, or newly created workspace.

*bstrEnhancementsFile*
Specifies the filename of the comma- or tab-delimited text file that lists the enhancements.

Because this file is parsed using the JET text Installable Sequential Access Module (ISAM), it cannot deviate from the following format:

```
GUID   Title   Show Reference Preload URL  Address   Expiration Date
```

Where each column contains the following data:

*GUID*
Specifies a unique identifier for the enhancement.

If you specify this parameter as an empty string (""), the **LoadEnhancementsFromFile** method automatically generates a unique identifier (GUID) for the enhancement. This option should only be used if the calling application has no need to further reference the enhancement entry.

*Title*
Specifies the title of the enhancement.

This value is displayed by TV Viewer when an episode has more than one enhancement available. When the user clicks the enhancement icon in the TV banner, TV Viewer displays a list of the enhancement titles, enabling the user to select which enhancement to display.

Example: "Fresh Prince Enhanced"

*Show Reference*

The show reference string for the broadcast that the enhancement applies to. You can obtain a show reference by calling either the **ITelevisionServices::MakeLocalBroadcastSchedule** or **ITelevisionServices::MakeRemoteBroadcastSchedule** methods.

If *bstrShowReference* contains an episode title, the enhancement is treated as an episode enhancement, and is only set for the episode(s) specified.

If the show reference contains an empty episode title (""), it is assumed to be a channel enhancement, and the **LoadEnhancement** and **LoadEnhancementsFromFile** methods search the tuning space for the union of the station, network, and channel number parameters specified in *bstrShowReference*. The method then sets an enhancement for all matching channel records.

Example: "1997/7/29!26673/7175/9690!0:30!0!0!0!0!9690!9690!4096!7040!"!'CBUT'!3!54!Fresh Prince of Bel-Air".

For an exact description of show reference string syntax, see Show Reference Format.

*Preload URL*

Specifies the URL of the HTML file that contains the Layout of the enhancement. If you do not specify a complete path, such as "C:\MyEnhance\", the URL is resolved relative to "C:\Program Files\TV Viewer\Layouts".

For example, if you specify *bstrPreloadURL* as "Cspan\Cspan.htm" it will be resolved to C:\Program Files\TV Viewer\Layouts\Cspan\Cspan.htm".

*Address*

A string that specifies the network card address, multicast address and port information used to connect to the enhancement stream. It must have the following format: "*netcard_address*\t*multicast_address*\t*multicast_port*".

Example: "125.125.125.125\t225.225.225.255\t10024"

*Expiration Date*

The date on which the enhancement expires.

All enhancements should have an expiration date. This enables Loadstub to delete obsolete enhancement data from the Guide database.

To locate more information about the Jet ISAM, see Further Information on Television Services for the Client.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

### Remarks

In addition to creating a properly-formatted text file, ISAM also requires you to include a schema file in the same directory. The schema file must be named Schema.ini and specify the name of the enhancement text file in its first line, enclosed in brackets ([ ]).

For example, if you wanted to load E:\Windows\Attach.txt, you must also create a file named Schema.ini in the e:\windows directory, with '[Attach.txt]' as the first line in the Schema.ini.

The following schema file is valid for comma-delimited files. Replace *Attachments.txt* with the name of your enhancement text file.

```
----------Schema.ini-----------
[Attachments.txt]
Format=CSVDelimited
ColNameHeader=False
CharacterSet=ANSI
Col1=ID Text
Col2=Title Text
Col3="Show Reference" Text
Col4="Preload URL" Text
Col5=Address Text
Col6="Expire Date" DateTime
```

The following schema file is valid for tab-delimited files. Replace *Attachments.txt* with the name of your enhancement text file.

```
-------Schema.ini--------
[Attachments.txt]
Format=TabDelimited
ColNameHeader=False
CharacterSet=ANSI
Col1=ID Text
Col2=Title Text
Col3="Show Reference" Text
Col4="Preload URL" Text
Col5=Address Text
Col6="Expire Date" DateTime
```

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

### See Also

**ITelevisionServices::LoadEnhancement**, **ITelevisionServices::DeleteEnhancementFromID**

# ITelevisionServices::DeleteEnhancementFromID

[This is preliminary documentation and subject to change.]

The **DeleteEnhancementFromID** method deletes the specified enhancement record from the Guide database. This method does not delete the corresponding HTML layout files.

```
HRESULT DeleteEnhancementFromID(
  Workspace *piDAOWorkspace,
  BSTR bstrEnhancementID
);
```

**Parameters**

*piDAOWorkspace*
A pointer to a DAO workspace. This can be either a preexisting, or newly created workspace.
*bstrEnhancementID*
The unique enhancement identifier.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

**Remarks**

In order to use the **DeleteEnhancementFromID** method to remove enhancement data from the Guide database, you must know the unique identifier of the enhancement. Typically this method is only called by the applications that originally loaded the enhancement data.

For example, a broadcast provider can send an delete announcement that causes the announcement filter to remove enhancement data from the database. In this case, a list of that broadcaster's enhancement identifiers is maintained at the head end and the appropriate identifier is sent with the delete announcement.

It is not necessary to use the **DeleteEnhancementFromID** method to delete obsolete enhancement data from the database. This is done periodically by Loadstub. To enable automatic deletion of enhancement data, you must set an expiration date for the enhancement data at the time that you load it into the Guide database.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

### See Also

**ITelevisionServices::LoadEnhancement**, **ITelevisionServices::LoadEnhancementsFromFile**

# ITelevisionServices::DeleteOldEnhancements

[This is preliminary documentation and subject to change.]

The **DeleteOldEnhancements** method deletes expired enhancement data from the Guide database. This method does not delete the corresponding HTML layout files.

Expired enhancements are those which have an expiration date, specified in the EN Expired Date field, that is earlier than the current date. The expiration date of enhancement data is set when the data is loaded into the Guide database, typically by using either **ITelevisionServices::LoadEnhancement** or **ITelevisionServices::LoadEnhancementsFromFile**.

```
HRESULT DeleteEnhancementFromID(
  Workspace *piDAOWorkspace,
);
```

### Parameters

*piDAOWorkspace*
A pointer to a DAO workspace. This can be either a preexisting, or newly created workspace.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

## ITelevisionServices::RemapEnhancements

[This is preliminary documentation and subject to change.]

The **RemapEnhancements** method maps enhancements loaded in the database to their corresponding channels and episodes, setting the appropriate values for the Enhancement Mapping ID field of the Channel and Episode tables in the Guide database.

```
HRESULT RemapEnhancements(
  Workspace *piDAOWorkspace,
);
```

### Parameters

*piDAOWorkspace*
Pointer to a DAO workspace. This can be either a preexisting, or newly created workspace.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

### Remarks

This method is called automatically by Loadstub whenever new Program Guide data is loaded into the Guide database. It is unlikely that your application will ever need to call this method explicitly.

As an example of the use of **RemapEnhancements**, suppose a content provider transmits enhancement data for a future episode. The enhancement filter stores the data in the Episode Enhancements table of the Guide database, even though the corresponding episode data is not yet loaded in the Guide database. However, when the program data is loaded, the new episodes are not matched to the previously loaded enhancement data. In other words, the E Enhancement Mapping ID is blank. To correct this, Loadstub calls **RemapEnhancements** to map the preexisting enhancements with the new episodes.

## IScheduledItems

[This is preliminary documentation and subject to change.]

The **ScheduledItems** collection enumerates the currently-scheduled show reminders. For more information, see Show Reminders.

**When to Implement**

**IScheduledItems** is implemented in Tssadmin.dll.

**When to Use**

Call the methods of **IScheduledItems** to retrieve, delete, or set show reminders.

**Methods in Vtable Order**

| **IUnknown Methods** | **Description** |
|---|---|
| **QueryInterface** | Returns pointers to supported interfaces |
| **AddRef** | Increments reference count |
| **Release** | Decrements reference count |

| **IScheduledItems** | **Description** |
|---|---|
| **Item** | Returns for a show reminder the specified show reference, application to start, working directory, time when advance notification should occur, and any optional command-line parameters. |
| **get_Count** | Returns the number of currently scheduled reminders in the collection. |
| **Add** | Schedules one or more broadcasts. |
| **Remove** | Deletes a scheduled show reminder. |
| **AddFromQuery** | Schedules reminders for all of the television shows in the result set of the specified query definition. |
| **get__NewEnum** | Returns an reference to an enumerator that iterates through a list of the scheduled reminders. |

**Remarks**

The **ScheduledItems** collection is part of the Television System Services (TSS) object library, tssadmin.dll. It contains the scheduled show reminders and has the usual methods of a collection interface: **get_Count**, **get__NewEnum**, and **Item**. The indices into the collection are strings that are the file names of tasks in Task Scheduler. TSS uses Task Scheduler for all scheduling. You obtain a reference to the **IScheduledItems** collection using the **ITelevisionServices::get_ScheduledItems** method.

**IScheduledItems** is derived from the **IDispatch** interface. To locate more information on **IDispatch**, a Component Object Model (COM) interface, see Further Information on Television Services for the

Client.

**See Also**

**ITelevisionServices**

# IScheduledItems::Add

[This is preliminary documentation and subject to change.]

The **Add** method schedules one or more show reminders.

```
HRESULT Add(
  BSTR   Workspace,       // in
  BSTR   ShowReference,   // in
  BSTR   Application,     // in
  BSTR   Directory,       // in
  unsigned long  AdvanceMinutes, // in
  BSTR   Parameters,      // in
  VARIANT * pSafeArray    // out
);
```

**Parameters**

*Workspace*
Name of a DAO workspace. This can be either a preexisting, or newly created workspace.

*ShowReference*
Show reference string. You can create a show reference by calling either **ITelevisionServices::MakeLocalBroadcastSchedule** or **ITelevisionServices::MakeRemoteBroadcastSchedule**.

*Application*
Name and path of the application that displays the reminder to the user when the scheduled show is on. Typically, TV Viewer is used to display reminders, and this value is set to "C:\Program Files\TV Viewer\Tvx.exe".

You can get the path to the client's installation of TV Viewer by calling **ITelevisionServices::get_DatabaseFile** and stripping "Tss.mdb" off of the end of the returned string. This path is also stored in this registry value:

**HKLM\Software\Microsoft\TV Services\ProductDir**

*Directory*
Name of the working directory that contains the reminder application.

*AdvanceMinutes*
Interval, in minutes, before the show's start that the reminder should occur.

*Parameters*

Optional command-line arguments to pass to the reminder application when it runs the reminder. The command-line syntax for these arguments is as follows:

*application* **/b** "*showreference*" **/u** "*username*" *otherarguments*

The following table lists and defines the possible values for these arguments.

| Placeholder | Description |
|---|---|
| *application* | The name and path of the application that displays the show reminder to the user. |
| *showreference* | The string denoting the show or shows being scheduled, in the show reference format. |
| *username* | The name of the owner of the DAO workspace used to schedule the reminder, typically "GuestUser".. |
| *otherarguments* | Additional command-line arguments for *application*. The format and number of these arguments depends on *application*. *otherarguments* is passed to *application* as command-line arguments.<br><br>For example, if you are using a custom application to display the reminder to the user, and this application has three styles of reminder dialog boxes, *otherarguments* could be used to specify the reminder style. |

*pSafeArray*

Pointer to a **VARIANT** that receives an **IEnumVARIANT** interface that enumerates the string indices of the scheduled tasks.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

**Remarks**

If the time and date specified in the *ShowReference* parameter are unambiguous, the Add method schedules a single show reminder.

Ambiguous show references can represent multiple or repeating broadcasts. For example, if you wanted to schedule a reminder for each weekly episode of a show, you could specify the time and channel of that show, leaving the date ambiguous. The show reference would then apply to all weekly episodes. In contrast, an unambiguous show reminder would specify the date, and thus apply to only a single episode.

If the show reference is ambiguous, the **Add** method uses the **ITelevisionServices::ResolveBroadcast** method to produce a query that matches *ShowReference*. In other words, a query for all episodes that match the ambiguous date or channel value. For example, if the show reference specified channel 27 at 5:00pm on Fridays, the generated query would return a listing of each episode in the Guide database that matched those parameters.

The **Add** method would then call the **IScheduledItems::AddFromQuery** method to schedule all of the rows returned by the query.

Reminders set using **IScheduledItems::Add** are not automatically visible in the the TV Viewer user interface. In order for a TSS-set reminder to be visible in TV Viewer it must meet certain standards. These standards are specified in Setting a Reminder that Appears in TV Viewer.

If the reminder you are adding is a record reminder, you should use the Task Scheduler to set the TASK_FLAG_SYSTEM_REQUIRED flag for the reminder. This causes TV Viewer to tune to the channel even if the system is sleeping. Otherwise, if the system is sleeping, TV Viewer will not wake up to run the record reminder.

In addition, if the record reminder has an application associated with it that automates tuning the VCR this application should be specified in the StartRecordingApp and/or EndRecordingApp values under this registry key:

**HKLM\Software\Microsoft\TV Services\Explorer\**

The TASK_FLAG_SYSTEM_REQUIRED flag should not be set for standard show reminders. Version 1.0 of Broadcast Architecture does not handle show reminders that go off while the system is sleeping.

For more information see Setting a Show Reminder and Setting a Record Reminder.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

**See Also**

**IScheduledItems::AddFromQuery**, **IScheduledItems::Remove**, **ITelevisionServices::ResolveBroadcast**

## IScheduledItems::AddFromQuery

[This is preliminary documentation and subject to change.]

The **AddFromQuery** method schedules reminders for all of the television shows in the result set of the **QueryDef** definition named by the *QueryName* parameter.

```
HRESULT AddFromQuery(
  BSTR  Workspace,      // in
  BSTR  QueryName,      // in
  BSTR  Application,    // in
  BSTR  Directory,      // in
  unsigned long  AdvanceMinutes, // in
  BSTR  Parameters,     // in
  VARIANT * pSafeArray  // out
);
```

### Parameters

*Workspace*
: Name of the DAO workspace. This can be either a preexisting, or newly created workspace.

*QueryName*
: Name of the **QueryDef** definition. Use the **ITelevisionServices::ResolveBroadcast** method to create a DAO **QueryDef** definition from a show reference.

*Application*
: Name and path of the application that displays the reminder to the user when the scheduled show is on. Typically, TV Viewer is used to display reminders, and this value is set to "C:\Program Files\TV Viewer\Tvx.exe".

  You can get the path to the client's installation of TV Viewer by calling **ITelevisionServices::get_DatabaseFile** and stripping "Tss.mdb" off of the end of the returned string. This path is also stored in this registry value:

  **HKLM\Software\Microsoft\TV Services\ProductDir**

*Directory*
: Name of the working directory for *Application*.

*AdvanceMinutes*
: Interval, in minutes, before the show's start that the reminder should occur.

*Parameters*
: Additional command-line arguments for *Application*. The format and number of these arguments depends on *Application*. *Parameters* is passed to *Application* as command-line arguments.

  For example, if you are using a custom application to display the reminder to the user, and this application has three styles of reminder dialog boxes, *Parameters* could be used to specify the reminder style.

*pSafeArray*
: Pointer to a **VARIANT** that receives an **IEnumVARIANT** interface that enumerates the string indices of the scheduled tasks.

### Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

### Remarks

The **AddFromQuery** method iterates through the result set of the defined query, composing a show reference for each row and scheduling a reminder for that show reference. Typically, you use the queries created by **ITelevisionServices::ResolveBroadcast** as the *QueryName* parameter for **AddFromQuery**.

Reminders set using **IScheduledItems::AddFromQuery** are not automatically visible in the the TV Viewer user interface. In order for a TSS-set reminder to be visible in TV Viewer it must meet certain standards. These standards are specified in Setting a Reminder that Appears in TV Viewer.

To locate more information on the **SAFEARRAY** data type, see Further Information on Television Services for the Client.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

### See Also

**IScheduledItems::Add**, **IScheduledItems::Remove**, **ITelevisionServices::ResolveBroadcast**

## IScheduledItems::get__NewEnum

[This is preliminary documentation and subject to change.]

The **get__NewEnum** method returns an reference to an enumerator that iterates through a list of the scheduled reminders.

```
HRESULT get__NewEnum(
  IUnknown ** ppUnk  // out
);
```

### Parameters

*ppUnk*

Reference to an enumerator that you can use to iterate through the scheduled reminders. The returned object implements the VARIANT enumerator interface, **IEnumVARIANT**.

To locate more information about enumerating collections using **IEnumVARIANT**, see Further Information on Television Services for the Client.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

**See Also**

**get_Count**, **Item**

# IScheduledItems::get_Count

[This is preliminary documentation and subject to change.]

The **get_Count** method returns the number of currently scheduled reminders in the **ScheduledItems** collection.

```
HRESULT get_Count(
  long * plRetVal  // out
);
```

**Parameters**

*plRetVal*

Pointer to a long that receives the number of reminders.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK.

Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

**See Also**

**get__NewEnum**, **Item**

# IScheduledItems::Item

[This is preliminary documentation and subject to change.]

The **Item** method returns for a show reminder the show reference, application to be started, working directory, advance notification interval, and any optional command-line parameters.

```
HRESULT Item(
  BSTR  Index,           // in
  BSTR * User,           // out
  BSTR * ShowReference,  // out
  BSTR * Application,    // out
  BSTR * Directory,      // out
  unsigned long * AdvanceMinutes,  // out
  BSTR * Parameters,     // out
  VARIANT * Task         // out
);
```

**Parameters**

*Index*
Index of the show reminder for which to retrieve information.

*User*
Pointer to a BSTR that receives the name of the owner of the DAO workspace used to schedule the reminder.

*ShowReference*
Pointer to a BSTR where this method returns the show reference string. For more information about the format of this string, see Show Reference Format.

*Application*
Pointer to a BSTR that receives the name and path of the application used to display the reminder to the user. Typically, TV Viewer is used to display reminders, and this value is set to "C:\Program Files\TV Viewer\Tvx.exe".

*Directory*

Pointer to a BSTR that receives the working directory of *Application*.

*AdvanceMinutes*

Pointer to a long that receives the interval, in minutes, before the show's start that the reminder should be displayed .

*Parameters*

Pointer to a BSTR that receives the additional command-line arguments set for *Application*. The format and number of these arguments depends on *Application*.

*Task*

Address where this method returns a **VARIANT** data type containing an **ITask** object, from which you can obtain the **ITaskTrigger** object. The **ITaskTrigger** object contains additional information about when and how the scheduled reminder will run.

**Return Values**

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

**Remarks**

The *Task* parameter is returned as a **VARIANT** instead of a Task object to enable Visual Basic programmers to safely use the **IScheduledItems::Item** method. Visual Basic cannot access the **ITask** interface; if this object was not returned as a VARIANT Visual Basic programmers would not be able to correctly release the Task object returned by **Item**.

If you are programming in C++, simply cast the **VARIANT** to an **ITask** interface variable.

If you are programming in Visual Basic, you should create a variable of type **Variant** and pass it as the *Task* parameter. Do not use this variable after you call the **Item** method. When the variable goes out of scope, Visual Basic releases the task object it references. It is recommended that you declare the **Variant** variable as local in a function that calls the **Item** method. Such a declaration causes the task object to be released quickly.

To locate more information on **ITask** and **VARIANT**, see Further Information on Television Services for the Client.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

**See Also**

**get__NewEnum**, **get_Count**

# IScheduledItems::Remove

[This is preliminary documentation and subject to change.]

The **Remove** method deletes a scheduled show reminder.

```
HRESULT Remove(
  BSTR  Index  // in
);
```

## Parameters

*Index*

Index of the show reminder to delete from the collection.

## Return Values

Returns an **HRESULT** indicating success or failure. If the method succeeds it returns S_OK. Otherwise it returns an error code. For specific error code values see Winerror.h and Dbdaoerr.h.

## Remarks

You can obtain the index for a scheduled show reminder from the enumerator method **IScheduledItems::get__NewEnum**, or as a return value from the **IScheduledItems::Add** or **IScheduledItems::AddFromQuery** method.

## QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in tssadmin.odl.
**Import Library:** Included as a resource in tssadmin.dll.

## See Also

**IScheduledItems::Add**, **IScheduledItems::AddFromQuery**, **IScheduledItems::get__NewEnum**

# Show References

[This is preliminary documentation and subject to change.]

To support the programmatic exchange of references to television shows, Television System Services (TSS) has a data format to specify television shows, called a *show reference*. Show references are strings that you can use to specify a single broadcast of a show, recurring broadcasts, or a time and a channel that can be matched to a broadcast at a later time. Show references also provide various other ways to specify shows, as described in the following sections.

You create a show reference by calling either the **ITelevisionServices::MakeLocalBroadcastSchedule** or **ITelevisionServices::MakeRemoteBroadcastSchedule** method of the **ITelevisionServices** interface, passing a task trigger object. To create a show reference for a recurring broadcast, you must pass an ambiguous trigger to the **ITelevisionServices** methods.

For more information, see the following topics:

- Show Reference Format
- Ambiguous Triggers

**Note** The description of the show reference format is for debugging purposes only. This format may change in future versions of Broadcast Architecture. Developers should compose and parse show references using the methods of the **ITelevisionServices** interface.

# Show Reference Format

[This is preliminary documentation and subject to change.]

A show reference is a null-terminated *ASCII* string containing fields delimited by exclamation marks ( ! ). Numerical fields are spelled out in ASCII digits.

The syntax of a show reference string is as follows:

```
yyyy/mm/dd!yyyy/mm/dd!hh:MM!du!in!fl!type!w!w!w!w!'network'!'station'!
'TuningSpace'!channel!title
```

The following table describes the placeholders in show reference syntax. All times are specified in coordinated universal time (UTC) in a 24-hour clock.

| Placeholder | Description |
|---|---|
| *yyyy* | The year. If this field contains the five digits 65535, it means any year. The first *yyyy* field in the syntax is for the start date; the second is for the end date. |
| *mm* | The month. If this field contains the digits 65535, it means any month. The first *mm* field is for the start date; the second is for the end date. |
| *dd* | The day of the month. If this field contains the digits 65535, it means any day. The first *dd* field is for the start date; the second is for the end date. |
| *network* | The network affiliate of the station. This affiliate can be the same as the station in some cases — for example, Home Box Office (HBO) or Cable News Network (CNN). In other situations, a national network such as Public Broadcasting System (PBS) is the affiliate for a local station. |
| *hh* | The hour portion of the time at which the show begins. If this field contains the digits 65535, it means any hour of the day. |
| *MM* | The minute portion of the time at which the show begins. If this field contains the digits 65535, it means any minute of the hour. |
| *du* | The duration of the show in minutes. This value is valid only if the TASK_TRIGGER_FLAG_KILL_AT_DURATION_END flag is set in the *fl* parameter of the show reference. |
| *in* | The interval at which to run the show reminder. Because a typical show reminder only runs once, this value is usually zero. |
| *fl* | Flags for interpretation of the preceding dates. These flags are the same as the ones in the **TASK_TRIGGER** structure of the Task Scheduler : TASK_TRIGGER_FLAG_HAS_END_DATE, TASK_TRIGGER_FLAG_KILL_AT_DURATION_END, and TASK_TRIGGER_FLAG_DISABLED. |
| *type* | This parameter is set as the **TriggerType** member of the **TASK_TRIGGER** structure. It takes a TASK_TRIGGER_TYPE data type. To locate more information on **TASK_TRIGGER**, see Further Information on Television Services for the Client. |
| *w* | This parameter is set as the **Type** member of the **TASK_TRIGGER** structure. It takes a TRIGGER_TYPE_UNION data type. |

| | |
|---|---|
| *station* | The station's call letters, for example KDKA. This field can be empty. |
| *TuningSpace* | A string that contains the name of the source, or tuning space, of the broadcast. An empty string (") in this field means any source. Tuning space names are strings, such as 'Cable', that represent a tunable video source, for example, cable, satellite broadcast, or terrestrial analog broadcast. Tuning space names are stored as named values under the \HKLM\Software\Microsoft\TVServices\Tuning Spaces\*TuningSpaceNumber* registry key where *TuningSpaceNumber* indicates the numerical identifier of the tuning space. |
| *channel* | An integer denoting the *channel* in the tuning space. If this field contains –1 (minus one), it means any channel. |
| *title* | The title of the television show. This title must not contain double quotation marks ( " ). The *title* field can be a limited regular expression with the syntax and semantics defined in Structured Query Language (SQL). Thus, "*" means any title, "*News*" means any title with "News" in it, and so on. To locate more information on SQL, see Further General Information. |

**Note** This description of the show reference format is for debugging purposes only. This format may change in future versions of Broadcast Architecture. Developers should compose and parse show references using the **MakeLocalBroadcastSchedule** and **MakeRemoteBroadcastSchedule** methods of the **ITelevisionServices** interface.

# Ambiguous Triggers

[This is preliminary documentation and subject to change.]

The Broadcast Architecture supports a custom implementation of the **ITaskTrigger** interface of Internet Client that enables an application to create ambiguous task trigger objects. An ambiguous trigger is one where the date and/or time parameters are not known at the time of the trigger's creation. You can also use the Broadcast Architecture implementation of **ITaskTrigger** to create a trigger where the time and date are fully specified.

You can use ambiguous triggers to represent a range of shows. For example, if a show appears every weeknight at 7 p.m., you can create a date-ambiguous trigger that represents all episodes of the show. For information on how to do this, see Creating an Ambiguous Trigger.

Once an ambiguous trigger is created, it can be passed to either the **ITelevisionServices::MakeLocalBroadcastSchedule** or **ITelevisionServices::MakeRemoteBroadcastSchedule** method to create a show reference that represents multiple episodes. You can then schedule a *show reminder* for each episode by calling the **IScheduledItems::Add** method and passing this multiepisode show reference.

The Broadcast Architecture implementation of **ITaskTrigger** implements both the **ITaskTrigger::SetTrigger** and the **ITaskTrigger::GetTrigger** methods. These are functionally equivalent to the methods implemented by the Task Scheduler. They differ only in that they support ambiguous triggers.

However, the Broadcast Architecture implementation does not support the **ITaskTrigger::GetTriggerString** method. Calling this method produces an error.

To locate more information about **ITaskTrigger**, see Further Information on Television Services for the Client.

## Creating an Ambiguous Trigger

[This is preliminary documentation and subject to change.]

To create an ambiguous trigger, first create a **TASK_TRIGGER** structure and set the ambiguous time or date members to –1. Then set this structure to a trigger object by calling the **ITaskTrigger::SetTrigger** method.

The following example creates an ambiguous trigger where the starting time, 7 p.m., is known but the day, month, and year are not. This trigger can be used, for example, to represent multiple episodes of a show that is broadcast every weeknight at 7 p.m.:

```
ITaskTrigger *pTaskTrig;
TASK_TRIGGER tt;
HRESULT hr;

memset(&tt, 0, sizeof(tt));
tt.cbTriggerSize = sizeof(TASK_TRIGGER);
tt.TriggerType = TASK_TIME_TRIGGER_ONCE;
tt.rgFlags = 0;
tt.wStartMinute = 0;
//Set the ambiguous time members
tt.wBeginYear = -1;
tt.wBeginMonth = -1;
tt.wBeginDay = -1;
//set the starting time to 7pm
tt.wStartHour = 19;

//Create an instance of a task trigger
if (FAILED(hr = CoCreateInstance(CLSID_TaskTrigger, NULL, CLSCTX_INPROC_SERVER, IID
  return hr;

//Set the ambiguous trigger in the task
if (FAILED(hr = pTaskTrigger->SetTrigger(&tt)))
{
  pTaskTrigger->Release();
```

```
    return hr;
  }
  else
  {
    //... Additional code that uses the trigger

    pTaskTrigger->Release();
    return NOERROR;
  }
```

You can set any number of the **TASK_SCHEDULER** structure members to –1. For example, you can make a trigger that is ambiguous only in the day, or the month, or the year. So, if you know that a show was on during January 1998 but don't know what day, you can set the month to 1, the year to 1998, and the day to –1.

To determine whether a trigger is ambiguous, you can call the **ITaskTrigger::GetTrigger** method and check whether any of the date or time members in the **TASK_TRIGGER** structure are set to –1.

To locate more information about **TASK_TRIGGER** and the **ITaskTrigger** interface, see Further Information on Television Services for the Client.

# Show Reminders

[This is preliminary documentation and subject to change.]

Show reminders are Task Scheduler tasks that remind a user to watch or record a television broadcast. Show reminders are set by TV Viewer. You can also write an application to set, modify, or delete show reminders.

For more information, see the following topics:

- About Show Reminders, which describes show reminder functionality and the format used to schedule a reminder in the Task Scheduler.
- Scheduling Show Reminders, which explains how to programmatically schedule a show reminder.
- **IScheduledItems**, which describes the **ScheduledItems** collection of scheduled show reminders.

# About Show Reminders

[This is preliminary documentation and subject to change.]

TV Viewer schedules show reminders as tasks in the Task Scheduler. These tasks specify the application to run, in this case TV Viewer, the command-line arguments that specify the user who set the reminder, and the show that the user requested the reminder for.

When the Task Scheduler runs a show reminder task, it sends these command-line parameters to TV Viewer or other specified application. If TV Viewer is not currently running, the Task Scheduler starts it. In either case, TV Viewer presents a dialog box to the user reminding the user of the show and asking whether TV Viewer should tune to the show broadcast.

If TV Viewer was not previously running and the user elects not to view the show, TV Viewer exits.

For more information on the parameters an application must set to create a show reminder, see Show Reminder Format.

# Show Reminder Format

[This is preliminary documentation and subject to change.]

You can use the following syntax to launch a show reminder, either from the command line or programmatically as a task in the Task Scheduler. The TV Viewer creates show reminders by setting the parameters shown in the **Run** property of tasks that it schedules in the Task Scheduler. For more information, see Scheduling Show Reminders.

### Show Reminder Syntax

The following shows the syntax for TV Viewer show reminders.

```
pathTvx.exe /b ShowReference /u User /a TVViewerParameters
```

### Show Reminder Parameters

The following lists and describes the parameters used in show reminder syntax.

*Path*

String that specifies the location of TV Viewer on the user's machine, for example C:\Program Files\TV Viewer\. When the reminder runs, TV Viewer will display the show reminder to the user.

*ShowReference*

String that contains a properly formatted show reference. For more information about the show reference format, see Show References.

You can create a show reference programmatically by calling either the **ITelevisionServices::MakeLocalBroadcastSchedule** or **ITelevisionServices::MakeRemoteBroadcastSchedule** method.

*User*

String containing the name of the user that set the reminder.

*TVViewerParameters*

String that contains additional parameters. This string should be formatted as follows:

**"tvx!***Type***!**[*ShowDuration***!**]**"**

*Type*

Flag value that specifies the type of reminder. The following table lists and describes the possible values for *Type*.

| Value | Meaning |
|---|---|
| Remind | A reminder to watch a show. This flag causes TV Viewer to remind a viewer a show is on. The viewer can then tune manually to the show. |
| Record | A reminder to record a show. This flag causes TV Viewer to tune to a show automatically. If the viewer sets a recording device to receive output at that time, the show is recorded. |

*ShowDuration*

Optional parameter that specifies the length of the show, in minutes.

Your application can obtain the TV Viewer path information programmatically by calling the **ITelevisionServices::get_DatabaseFile** method to find the location of the database, which is installed in the same location as the TV Viewer executable file, Tvx.exe. Your application must then remove the database file name from the string that the **ITelevisionServices::get_DatabaseFile** method returns.

## Show Reminder Example

The following example demonstrates how to run TV Viewer from the command line. Note that there are quotation marks around the path and file name. The quotation marks are necessary because the directory, Program Files, contains a space character.

```
"C:\Program Files\TV Viewer\Tvx.exe" /b "1997/4/22!73/1/0
!2:0!0!0!0!0!0!0!0!0!''!'LIFE'!'Cable'!38!Intimate Portrait"
/u "GuestUser" /a "tvx!Remind!60!"
```

# Overview of Program Guide Services

[This is preliminary documentation and subject to change.]

The Program Guide provided by Broadcast Architecture displays television and supports data services that broadcast television schedule information to the user. In addition, the user can use the Program Guide to tune to a specific show in the schedule and schedule reminders to watch a show.

This functionality is exposed programmatically by a set of software extensions known as Television System Services (TSS) that provides:

- Password security.
- Restrictions on unauthorized viewing.
- Interapplication scheduling of television shows for viewing or recording.

You use Television System Services for scheduling and control of television shows in conjunction with the Program Guide. Various broadcast client components can use Television System Services, including the Microsoft® ActiveX™ control for video (the Video control) and its underlying Microsoft® DirectShow™ filters and the Video Access server.

The TSS application programming interface (API) is available through Automation, as methods and properties of the **ITelevisionServices** interface and the supporting **IScheduledItems** interface.

The Program Guide maintains current information about program schedules in a Microsoft® Jet database, Tss.mdb. Because broadcast schedule information constantly changes, the Guide database must acquire and update broadcast data on a regular basis. This acquisition and update is done using a loader application.

The following topics examine various Program Guide services:

- Updating Guide Data describes the various problems involved in integrating Program Guide information from multiple networks and how dynamic-link libraries (DLLs) can be used to load data into the database supporting the Program Guide.
- Loader Libraries outlines the tasks that any Guide database loader must complete to operate successfully on the broadcast client. This section also includes details about how to write a custom loader application.
- Guide Data Objects covers a set of objects that applications use to retrieve or set values in the Guide database.
- Guide Database Schema provides a detailed description of the tables and fields in the Guide database.
- Program Guide Registry Entries defines the entries created in the system registry for *tuning spaces* and their associated loader applications.

# Updating Guide Data

[This is preliminary documentation and subject to change.]

As viewers tune to various channels on various broadcast sources, they expect the Program Guide to give them current information about programs being broadcast from all sources available to them. Meeting this information need poses several challenges.

For more information on how this need is met, see the following topics:

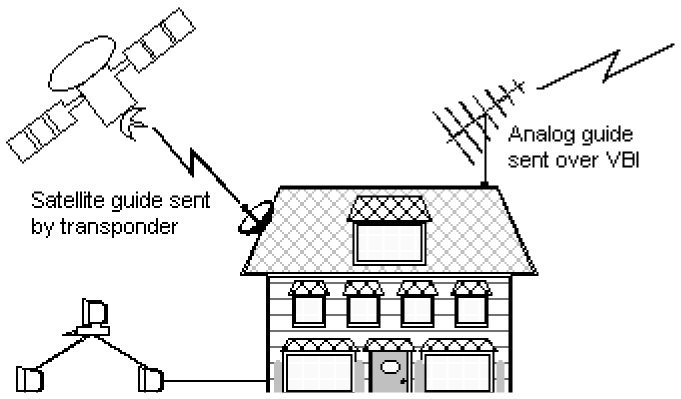
- Multiple Data Sources describes possible sources of Program Guide data.
- Overlapping Channels explains how to integrate Program Guide data from multiple and possibly conflicting data sources into a single data set.
- Moving Data to the Program Guide discusses dynamic-link libraries (DLLs) used by the loader application to map various Program Guide data formats to the single model used by the broadcast client.
- Updating the Guide Database describes the general process by which a database loader DLL moves Program Guide data into the Broadcast Architecture database.

## Multiple Data Sources

[This is preliminary documentation and subject to change.]

The Program Guide can obtain program data from various suppliers and in various formats. A satellite service provider may supply Program Guide data over one of the provided satellite channels. A local cable operator may supply Program Guide data through a vendor who sells subscriptions to such data downloaded from the Internet nightly. Satellite broadcast schedules may arrive as data packets from the satellite, and cable TV program data may arrive as a downloaded file or data transmitted over the *vertical blanking interval* (VBI) in the analog television signal.

The following illustration shows some of the various possible data sources feeding the Guide database.


Analog guide sent over VBI
Satellite guide sent by transponder

## Overlapping Channels

[This is preliminary documentation and subject to change.]

The Program Guide may have to combine information about multiple broadcast sources. For example, Program Guide data from the local cable company may have to reside in the Program Guide along with Program Guide data from a satellite service provider. This potential requirement can be problematic when channels from different systems use the same channel numbers.

To deal with issues arising from overlapping channel numbers, Broadcast Architecture implements the idea of tuning spaces. A *tuning space* is a set of nonoverlapping channels that are all available through the same type of physical channel *tuner*, such as an analog cable tuner. A broadcast client with multiple tuning devices may provide channels from multiple tuning spaces.

A physical device may support more than one tuning space. For example, if someone moves an analog receiver from San Francisco to Seattle, that person finds that TV broadcast channel 3 is different in the two cities. A viewer with both a cable connection and a conventional broadcast antenna, using a switch box to select between providers, has inputs that represent two tuning spaces.

The Program Guide must be able to handle and display information for more than one tuning space.

In addition, a channel in a single tuning space can be shared by multiple broadcast content providers. For example, channel 21 might broadcast network 1 from 9 p.m. to 9 a.m., and network 2 from 9 a.m. to 9 p.m. The Program Guide must also be able to handle and display such shared channels.

## Moving Data to the Program Guide

[This is preliminary documentation and subject to change.]

For any given broadcast service, one or more sources may provide Program Guide data. Because of the number of sources and formats for Program Guide data, it would be unwieldy for the Program Guide to directly receive Program Guide data. There are simply too many variations.

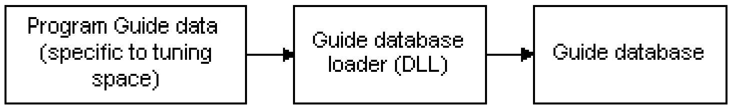
To maintain current information, new data must be uploaded to the Guide database. The job of updating the database belongs to a set of Guide database loaders. Each tuning space has an associated Guide database loader.

A Guide database loader may process multiple tuning spaces. For example, a database loader for analog networks may process data for both cable and over-the-air broadcasts.

A database loader:

- Accesses available sources of Program Guide data.
- Receives new data.
- Maps data to tables and fields in the Guide database.
- Loads the new data into the Guide database.

The following illustration shows the relationship between a database loader and the Guide database.


Program Guide data (specific to tuning space)
Guide database loader (DLL)
Guide database

# Updating the Guide Database

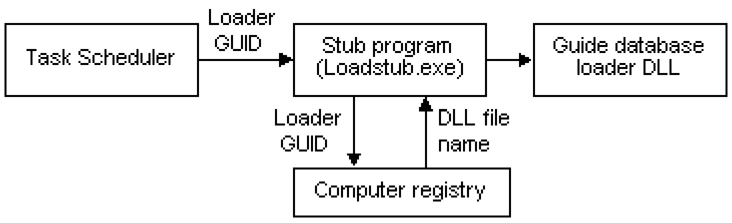
[This is preliminary documentation and subject to change.]

The process of loading current information into the Guide database begins with the Loadstub component. This component calls the loader library specified in its arguments to load data into the Guide database.

Typically, you set the Loadstub component as a task in the Task Scheduler. Making this component a task causes the Program Guide data for the tuning space associated with the specified loader to be automatically updated at regular intervals. The task for a particular tuning space specifies the Loadstub component that should be run and the globally unique identifier (GUID) of the appropriate database loader DLL.

At the designated time, the Task Scheduler runs the stub program (Loadstub.exe), passing the GUID of the Guide database loader DLL along with other command options. The stub program reads entries in the computer registry to identify the location and file name of the loader DLL, based on the DLL's specified GUID.

The following illustration shows how the system starts the loader DLL.



For details about the information that needs to be in the registry to support tuning spaces and Guide database loaders, see Program Guide Registry Entries.

When an application needs extensions to the Guide database, it must add any new fields or tables to the database during installation. Providers must design any software that updates the database to safely ignore references to fields or tables that are not present or that have an unclear purpose.

# Loader Libraries

[This is preliminary documentation and subject to change.]

For each tuning space, a corresponding Guide database loader exists. A database loader typically:

- Accesses the Guide database.
- Acquires Program Guide data from a specified source.
- Maps data from the Program Guide source to the appropriate fields in the Guide database.
- Updates the database with the new data.

Guide database loaders are DLLs. The specifics of creating a Guide database loader vary depending on:

- Tuning spaces supported.
- Source of the Program Guide data (for example, downloaded from the Internet versus broadcast live).
- Format of the Program Guide data.
- Correlation between the provided data and the Guide database schema.

Updating the Guide database involves accessing the Microsoft® Jet database engine from the DLL. For more information on creating and working with loaders, see the following topics:

- Loaders Provided by Broadcast Architecture
- Writing a Custom Loader

To locate information on accessing a Jet database, see Further General Information.

# Loaders Provided by Broadcast Architecture

[This is preliminary documentation and subject to change.]

Broadcast Architecture currently provides a loader that adds StarSight programming data to the Guide database. This loader operates on a data file.

For this loader, a data location may be stored in an entry under the loader's registry key. This data location is the name of the file to use for input, for example C:\ProgramFiles\Program GuideData\ss.bin.

The data file created for the StarSight loader can come by a number of means, for example the television tuner signal or Internet channel broadcasting.

# Writing a Custom Loader

[This is preliminary documentation and subject to change.]

Broadcast Architecture provides a sample loader framework that you can use to quickly develop a custom loader. This sample loader, Load.cpp, implements a class **CLoadApp** that contains all of the functionality of a database loader. It demonstrates the use of the Guide data objects to create a new show and schedule it. This set of objects provides a programmatic interface to the data in the Guide database, using which you can easily create records, delete records, retrieve field values, and set field values. To locate the sample loader, see Broadcast Architecture Sample Applications.

To make your own loader based on the sample, all you need to do is modify the **CLoadApp::Handle** method to load your data into the Guide database using the Guide data objects.

Even if you do not use the provided loader framework, it is recommended that you use the Guide data objects. These objects are fully tested and debugged and save you from having to reimplement their functionality.

## Loader Responsibilities

[This is preliminary documentation and subject to change.]

Because your loader DLL runs in the context of Broadcast Architecture, it should implement the following functionality:

- Sending loader event notifications
- Performing tuning space maintenance
- Processing quit events

In addition to the preceding, it is recommended that your loader DLL, if appropriate, support the **/I:***data_location* command-line argument. This argument enables users to override the location of Program Guide data in the registry. For more information, see Loadstub Command-line Parameters.

### Loader Event Notifications

[This is preliminary documentation and subject to change.]

Your loader DLL should send broadcast window messages when loader events occur. These messages can be sent using either the **SendMessageTimeout** or **PostMessage** function. To locate more information about the **SendMessageTimeout** and **PostMessage** functions, see Further Information on Program Guide Services for the Client.

The following table lists and describes these messages and their corresponding events.

| **Message** | **Event** |
|---|---|
| EPGLDR_STARTING | The loader starts to run. This message is sent by the Loadstub component and does not need to be sent by your loader DLL. |
| EPGLDR_ENDING | The loader finishes. This message is sent by the Loadstub component and does not need to be sent by your loader DLL. |
| EPGLDR_EXCLUSIVE_START | The loader is beginning an operation that requires exclusive access to the Guide database. This message is sent by the Loadstub component and does not need to be sent by your loader DLL. |
| EPGLDR_ EXCLUSIVE _END | The loader has finished an operation that requires exclusive access to the Guide database. This message is sent by the Loadstub component and does not need to be sent by your loader DLL. |
| EPGLDR_ACTIVE_COMMIT_START | The loader is beginning to update records. Applications that cache Program Guide data will need to refresh their queries. |
| EPGLDR_ACTIVE_COMMIT_END | The loader has finished updating records. Applications that cache Program Guide data should refresh their queries. |
| EPGLDR_PASSIVE_COMMIT_START | The loader is beginning to update records. Applications that cache Program Guide data do not need to refresh their queries. |
| EPGLDR_PASSIVE_COMMIT_END | The loader has finished updating records. Applications that cache guide data do not need to refresh their queries. |

**Note** A loader DLL should use the **SendMessageTimeout** method rather than **PostMessage** to send EPGLDR_ACTIVE_COMMIT_START and EPGLDR_EXCLUSIVE_START messages. For EPGLDR_ACTIVE_COMMIT_START, this is because **PostMessage** can cause the message to arrive at applications monitoring events after updating has begun. This message's arrival after updating starts may result in an application attempting to query the database while updating is taking place. For EPGLDR_EXCLUSIVE_START, the loader needs to work on the assumption that all other applications have released themselves from the Guide database before the loader continues with its exclusive operation (which may include database repair, compaction, and so on).

## Tuning Space Maintenance

[This is preliminary documentation and subject to change.]

Each loader DLL should clean up its corresponding tuning space. This cleanup consists of the following actions:

- Deletion of dangling references in the Channel, Episode, Station and Theme tables.
- Deletion of expired records from the Time Slot table. These are defined as records whose end time is more than one day in the past.
- Deletion of duplicate time slots from the Time Slot table. For example, if a show is listed as broadcasting on a particular channel from 10 a.m. to 12 p.m., and the loader adds a new time slot for that show from 10:30 to 11:30 a.m., one of time slot entries should be removed.

The preceding actions can be performed by running the cleanup queries defined in the Guide database. For more information, see Guide Database Query Reference.

## Guide Database Query Reference

[This is preliminary documentation and subject to change.]

The Guide database contains several queries that your application can run in order to remove unused or out-of-date data records. These queries can be run programmatically from the loader library by using the **CLoadApp::ExecuteActionQuery** method. This method is implemented in the sample loader provided with Broadcast Architecture. For more information, see Writing a Custom Loader.

The Guide database contains the following cleanup queries.

| Query | Action |
|---|---|
| Delete Dangling Channel | Deletes all records from the Channel table that do not have a matching entry in the Time Slot table. |
| Delete Dangling Episode | Deletes all records from the Episode table that do not have a matching entry in the Time Slot table. |
| Delete Dangling Station | Deletes all records from the Station table that do not have a matching entry in the Channel table. |

| | |
|---|---|
| Delete Dangling Theme | Deletes all records from the Theme table that do not have a matching entry in the Episode table. |
| Delete Omitted Time Slot | Deletes duplicate time slots. The query deletes the Time Slot with an older Last Updated time. For efficiency, pass the start time and end time of the period for which you want to eliminate duplicates, and also the tuning space. |
| Delete Expired Time Slot | Deletes all time slots with end times older than the specified time. |

## Processing Quit Events

[This is preliminary documentation and subject to change.]

When the loader library is loading data into the Guide database using the **CLoadApp::Handle** method, the loader is no longer in the operating system message loop. Thus, there is no way for the loader to receive a quit processing notification.

Instead, the Loadstub component monitors the operating system message queue for quit messages. At convenient times in loader processing, for example between adding records or executing queries, your loader should call the `pfnForceQuit` function that Loadstub.exe passes to the loader DLL. This function enables Loadstub to forward quit event notifications to the loader so that the loader can stop processing, if necessary.

Your loader can implement this functionality by calling the following code whenever loader processing can be gracefully stopped

```
if ((*pfnForceQuit()){
  //Code to clean up and gracefully exit the loader
}
```

where `pfnForceQuit` was passed at the entry point.

# Implementing the Entry-Point Function

[This is preliminary documentation and subject to change.]

A database loader DLL must implement a standard entry point to be called by Loadstub. The call to that entry point starts the DLL code that loads Program Guide data into the database. The entry point function must be named **Program Guide_DBLoad**.

The prototype for this function is

```
Typedef BOOL (*PFNFORCEQUIT)(VOID);
APIENTRY Program Guide_DBLoad(int &argc, _TCHAR **argv, CdbEngine &db, PFNFORCEQUIT
```

where the first two parameters provide support for standard command line processing, the next parameter is a reference to the database, and the final parameter is the quit processing callback function.

**Note** The sample loader, Load.cpp, already contains an implementation of **Program Guide_DBLoad**. Thus, if you build your loader using this framework, you do not need to implement this method.

# Loadstub

[This is preliminary documentation and subject to change.]

The Loadstub component, Loadstub.exe, starts a loader DLL. Typically, Loadstub is automatically started as a task in the Task Scheduler in the Microsoft® Windows® 98 operating system. Using a stub application provides a common interface for loader DLLs that themselves may be quite divergent in approach.

For more information, see the following topics:

- Running Loadstub, which describes how to start Loadstub and details what it does.
- Loadstub Command-line Parameters, which details the command-line parameters you can set for Loadstub.

## Running Loadstub

[This is preliminary documentation and subject to change.]

Guide database loaders are all called as needed by a standard loader stub program. This stub program, Loadstub.exe, is scheduled to run at appropriate times to update the Guide database. For more information on Loadstub options, see Loadstub Command-Line Parameters.

When the stub program starts, it:

1. Parses the command line for recognized options:

   1. Executes the **/X** command line option, if it exists.
   2. Starts the Microsoft® Jet engine.
   3. Executes the **/R** command line option to repair the database, if this option exists, and does a quick check and repair of corruption errors if it does not.
   4. Executes the **/C** command line option to compact the database, if the option exists. If there is no **/R** option, the program also repairs the database before compaction.
   5. Executes the **/LEF** command line option to load the specified enhancements file, if the option exists.
2. Uses option information to extract the name of the loader DLL from the registry.
3. Extracts user and password information.
4. Opens a workspace as `LoaderWSP` and a user as `loader`.
5. Creates a window message by calling the **RegisterWindowMessage** function. On the **RegisterWindowMessage** call, Loadstub passes the string representation of the loader's globally unique identifier (GUID).
6. Broadcasts an EPGLDR_STARTING message to notify client applications that the database is receiving updated data.
7. Starts the loader DLL.
8. Broadcasts an EPGLDR_ENDING message to notify client applications that the database update is completed and to indicate success or failure.
9. Updates the **LastAttempt** and **LastRun** entries in the **Program Guide Loaders** key of the system registry.

## Loadstub Command-Line Parameters

[This is preliminary documentation and subject to change.]

The Loadstub component is typically run as a task by the Task Scheduler. When this is the case, you specify the following syntax in the Run field of a scheduled task.

*Path***Loadstub.exe /L:***Loader_GUID* [*OptionalArguments*]

**Parameters**

*Path*
  Path to the Loadstub.exe component. If the path contains spaces, the path and file name must be surrounded by quotation marks ("), for example **"C:\Program Files\TV Viewer\Loadstub.exe"**

*Loader_GUID*
  GUID of the loader component. This parameter is required.

*Optional Arguments*
  Additional optional arguments. These can be any combination of the following.

| Argument | Description |
|---|---|
| **/C** | Specifies to repair and compact the database. |
| **/R** | Specifies to repair the database. |
| **/X** | Specifies to replace the database. The replacement file is specified by the registry entry **DBReplacementFile**, right next to the current **DBFile** entry. Specifying **DBReplacementFile** enables an application to replace a database programmatically. |
| **/P** | Specifies a partial update. This option allows a loader to implement a "quick" mode and a normal mode. For example, in partial update mode, a loader might only gather the next four hours of guide data, rather than the next two and a half days.<br><br>Note: Because the data format varies between loader libraries, this parameter is not handled by the Loadstub component. Instead, Loadstub passes the parameter as an argument to the loader. Thus, in order for this parameter to work, it must be supported by the loader. |
| **/I:***data_location* | Specifies the location of the data file. If this parameter is not set, Loadstub uses the location stored in the registry. For the StarSight loader, this parameter specifies the name of the data file.<br><br>Note: Because the type of data passed into this command-line parameter varies, this parameter is not handled by the Loadstub component. Instead, Loadstub passes the parameter as an argument to the loader. Thus, in order for this parameter to work, it must be supported by the loader. |
| **/LEF** *filename* | Loads the named enhancement file. |

**Examples**

The following example runs the StarSight loader, loading data from C:\Windows\Temporary Internet Files\ssdata.bin instead of the location specified in the registry.

```
"C:\Program Files\TV Viewer\LOADSTUB.EXE" /L:{C94D1940-9F69-11d0-BDB8-0000F8027346}
```

Note that because the path to the data contains spaces, it is surrounded by quotation marks. You can also use the short directory names, as shown following:

```
C:\Progra~1\TV Viewer\LOADSTUB.EXE /L:{C94D1940-9F69-11d0-BDB8-0000F8027346} /P /I:
```

# Guide Data Objects

[This is preliminary documentation and subject to change.]

The Guide Data objects are a set of objects that applications can use to retrieve or set values in the Guide database. They wrap the Data Access Object (DAO) code necessary to access the Guide database's records and fields.

It is recommended that your loader application use the Guide Data objects because they provide a convenient and tested programmatic interface to the records and fields of the Guide database.

These objects are also used internally by the TV Viewer application and the database loader applications installed with the Broadcast Architecture.

For more information, see the following topics:

- Guide Data Object Inheritance, which describes the architecture and inheritance of the Guide Data objects.
- Using the Guide Data Objects, which explains how to use the Guide Data objects to manipulate the records and fields in the Guide database.
- Guide Data Object Reference, which provides detailed descriptions of the methods exposed by the Guide Data objects.

To compile programs that utilize the Guide Data Objects, the Dbsets.h header file must be in the include directory of your C/C++ compiler. You must also include header files for each recordset object that you use. For example, Network.h defines the **CNetworkRecordset** and **CNetwork** objects. In addition, you should link to Dbsets.dll.

# Guide Database Schema

[This is preliminary documentation and subject to change.]

The collection of data describing current and upcoming broadcast video events is stored in a Microsoft® Jet database file called the Guide database. Broadcast Architecture uses a Jet database to take advantage of a widely accepted database format that is easily programmed from various development tools, including the Microsoft® Visual C++® development system and the Microsoft® Visual Basic® programming system. The Jet database engine provides several advantages. It:

- Provides a high-level interface, called Data Access Objects (DAO), supported by various versions of Visual Basic and Visual C++.
- Supports Structured Query Language (SQL) for easy definition of complex queries.
- Is an Automation server.

The Guide database schema contains the following tables.

| Table | Description |
|---|---|
| AdTrack | Stores information about rotating advertisements. |
| Broadcast Property | Gives names for various broadcast properties, such as closed captioning |
| Channel | Defines for each tuning space all properties and all channels that can be tuned to |
| Channel Property | Specifies properties associated with individual channels |
| Channel Stream | Lists the data streams available on a specific channel |
| Enhancement | Describes the links between enhancements and the show to be enhanced |
| Episode | Describes each television program episode |
| Episode Property | Specifies properties associated with individual episodes |
| Genre | Provides names for categories of episodes |
| Network | Defines various broadcast networks, such as ABC, CBS, NBC, and so on |
| Rating | Defines names for individual ratings used in various rating systems |
| Rating System | Identifies various rating systems, such as the Motion Picture Association of America (MPAA) system |
| Station | Defines individual broadcast providers |

| | |
|---|---|
| Stream Type | Gives names for all data streams available on a specific channel |
| Sub-Genre | Provides names for subcategories within a genre |
| Theme | Defines unique pairs of genres and subgenres to help classify episodes |
| Theme ID Mapping | Maps genres and subgenres defined by broadcast content providers to standard genre-subgenre pairs in the Theme table |
| Time Slot | Defines time slot and channel information for each episode |

For more information on how to examine the Guide database schema, see Viewing the Schema.

# Viewing the Schema

[This is preliminary documentation and subject to change.]

To investigate and print out the details of the Guide database schema, including field descriptions, field sizes, and referential integrity rules, you can make a copy of the empty database installed by the broadcast client installation program. You can then examine the copy using Microsoft® Access, as described in the following procedure.

**To examine the Guide database schema in detail**

1. Open Microsoft® Windows® Explorer.
2. Open the TV Viewer folder within the folder where the broadcast client software resides on your computer.
3. Select the two database files, Tss.mdb and Tss.mdw.
4. Copy these files to a different folder convenient to your development project.
5. Use Access to open these files, and explore and print out the database structure as explained in the Access documentation.

For improved performance and ease of navigation, many of the tables in the Guide database schema are referentially linked with automatically generated counters. The referential integrity of these counter fields is guaranteed by:

- The Microsoft® Jet database engine.
- The schema structure.
- The behavior of the database loader.

You can use these automatically generated identifier (ID) fields in queries and code that navigate among the tables. However, you should never store these identifier fields externally to the database, because their values are subject to ongoing change when the database loaders add and delete records as part of updating the database. If an application must save an external reference to an item in the database, it must use one of the key fields flagged as external to do so.

## AdTrack Table

[This is preliminary documentation and subject to change.]

The AdTrack table in the Guide database contains information about rotation ads. The AdTrack table contains the following fields.

| Field | Description |
|---|---|
| AdName | Name of the advertisement. |
| Impressions | Number of times the advertisement appears. |
| PageGroup | File name for the advertisement schedule. |

### AdName

[This is preliminary documentation and subject to change.]

The AdName field provides the name of the advertisement. This field's values are as follows:

Key: Not applicable
Index: Yes (duplicates okay)
Data type: Text
Field size: 20 characters
Default value: Not applicable
Required: Yes

### Impressions

[This is preliminary documentation and subject to change.]

The Impressions field provides the number of times, or impressions, that the advertisement is displayed to the user. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## PageGroup

[This is preliminary documentation and subject to change.]

The PageGroup field provides the name of the advertisement schedule file. The Program Guide uses the advertisement schedule file to determine what advertisements to show. This field's values are as follows:

Key: Not applicable
Index: Yes (duplicates okay)
Data type: Text
Field size: 20 characters
Default value: Not applicable
Required: Yes

# Broadcast Property Table

[This is preliminary documentation and subject to change.]

The Broadcast Property table in the Guide database gives names for various broadcast properties, such as closed captioning. The Broadcast Property table contains the following fields.

| Field | Description |
|---|---|
| BP Abbreviation | Short name for this property. |
| BP Broadcast Property ID | Unique identifier for this broadcast property record. |
| BP Display Order | Priority of this item when there is not room to display all properties. |
| BP Name | Name for this property. |
| BP Pictogram Moniker | File name of the bitmap associated with this property. |

| | |
|---|---|
| BP Tuning Space | Tuning space to which this broadcast property belongs. |

## BP Abbreviation

[This is preliminary documentation and subject to change.]

The BP Abbreviation field contains the short name for this property, restricted to 4 bytes in length (for example, cc, st, or ppv). This field's values are as follows:

Key: Not applicable
Index: No
Data type: Text
Field size: 4 characters
Default value: Zero-length string
Required: Yes

## BP Broadcast Property ID

[This is preliminary documentation and subject to change.]

The BP Broadcast Property ID field contains the unique identifier for this broadcast property record. This field's values are as follows:

Key: Primary
Index: Yes (no duplicates)
Data type: AutoNumber
Field size: Long integer
Default value: Automatically incremented
Required: Yes

## BP Display Order

[This is preliminary documentation and subject to change.]

The BP Display Order field identifies the priority of this item when there is not room to display all properties. This field's values are as follows:

Key: Not applicable
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## BP Name

[This is preliminary documentation and subject to change.]

The BP Name field contains the name for this property (for example, closed captioned, stereo, or pay-per-view). This field's values are as follows:

Key: Not applicable
Index: No
Data type: Text
Field size: 128 characters
Default value: Zero-length string
Required: No

## BP Pictogram Moniker

[This is preliminary documentation and subject to change.]

The BP Pictogram Moniker field contains the file name of the bitmap associated with this property. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Text
Field size: 255 characters
Default value: Zero-length string
Required: No

## BP Tuning Space

[This is preliminary documentation and subject to change.]

The BP Tuning Space field identifies the tuning space to which this broadcast property belongs. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Number
Field size: Long integer
Default value: Zero
Required: Yes

## Channel Table

[This is preliminary documentation and subject to change.]

The Channel table in the Guide database defines for each tuning space all properties and all channels that can be tuned to. The Channel table contains the following fields.

| **Field** | **Description** |
|---|---|
| C Channel ID | Internally generated unique identifier for this channel. |
| C Channel Number | Tuning identifier for the channel's input stream. |
| C Description | Extended description of this channel. |
| C Display Mask | Reserved for future use. |
| C End Time | Time when this channel becomes unavailable on the network. |
| C Enhancement Mapping ID | Mapping identifier from the Enhancement table. |
| C Last Update | Last time this row in the table was updated. |
| C Length | Length of the time the channel is available, in minutes. |
| C Payment Address | Reserved for future use. |
| C Payment Token | Reserved for future use. |
| C Rating ID | Foreign key from the Rating table. |
| C Start Time | Time when this channel becomes available on the network. |

| | |
|---|---|
| C Station ID | Station mapped to this channel during the specified time period. |
| C Tunable | Value that specifies whether the channel can be tuned to. |
| C Tuning Space | Logical identifier of the network type for the physical broadcast (for example, broadcast tuner, satellite, and so on). |

## C Channel ID

[This is preliminary documentation and subject to change.]

The C Channel ID field provides an internally generated unique identifier for this channel. This field's values are as follows:

Key: Primary
Index: Yes (no duplicates)
Data type: AutoNumber
Field size: Long integer
Default value: Automatically incremented
Required: Yes

## C Channel Number

[This is preliminary documentation and subject to change.]

The C Channel Number field provides the tuning identifier for the channel's input stream. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Number
Field size: Long integer
Default value: Zero
Required: Yes

## C Description

[This is preliminary documentation and subject to change.]

The C Description field provides an extended description of this channel. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Memo
Field size: Not applicable
Default value: Zero-length string
Required: No

## C Display Mask

[This is preliminary documentation and subject to change.]

Reserved for future use.

## C End Time

[This is preliminary documentation and subject to change.]

The C End Time field provides the time when this channel becomes unavailable on the network. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Date/time
Field size: Not applicable
Default value: NULL date
Required: No

## C Enhancement Mapping ID

[This is preliminary documentation and subject to change.]

The C Enhancement Mapping ID field provides the identifier of the enhancement in the Enhancement Mapping table. This field's values are as follows:

Key: Foreign key
Index: No
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## C Last Update

[This is preliminary documentation and subject to change.]

The C Last Update field identifies the last time this row in the table was updated. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Date/time
Field size: Not applicable
Default value: Not applicable
Required: No

## C Length

[This is preliminary documentation and subject to change.]

The C Length field provides the length of the time the channel is available, in minutes. This value is useful when channels are available only during certain times. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## C Payment Address

[This is preliminary documentation and subject to change.]

Reserved for future use.

## C Payment Token

[This is preliminary documentation and subject to change.]

Reserved for future use.

## C Rating ID

[This is preliminary documentation and subject to change.]

The C Rating ID field is a foreign key from the Rating table. This field's values are as follows:

Key: Foreign
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## C Start Time

[This is preliminary documentation and subject to change.]

The C Start Time field provides the time when this channel becomes available on the network. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Date/time
Field size: Not applicable
Default value: NULL date

Required: No

## C Station ID

[This is preliminary documentation and subject to change.]

The C Station ID field provides the station mapped to this channel during the specified time period. This field's values are as follows:

Key: Foreign
Index: Yes (duplicates okay)
Data type: Number
Field size: Not applicable
Default value: Zero
Required: Yes

## C Tunable

[This is preliminary documentation and subject to change.]

The C Tunable field specifies whether the channel represents a broadcast that can be tuned to. For example, users cannot tune to some Program Guide channels.

This field enables applications to quickly determine whether a channel can be tuned to without having to query the Channel Property table. This field is provided to enhance performance, as are the property fields in the Episode table.

This field's values are as follows:

Key: Not applicable
Index: Yes (duplicates okay)
Data type: Yes/no
Field size: Not applicable
Default value: Yes
Required: No

## C Tuning Space

[This is preliminary documentation and subject to change.]

The C Tuning Space field provides the logical identifier of the network type of the physical broadcast (for example, broadcast tuner, satellite, and so on). This field's values are as follows:

Key: Not applicable
Index: No
Data type: Number
Field size: Long integer
Default value: Zero
Required: Yes

# Channel Property Table

[This is preliminary documentation and subject to change.]

The Channel Property table in the Guide database specifies properties associated with individual channels. The Channel Property table contains the following fields.

| **Field** | **Description** |
|---|---|
| CP Broadcast Property ID | Foreign key from the Broadcast Property table. |
| CP Channel ID | Foreign key from the Channel table. |

## CP Broadcast Property ID

[This is preliminary documentation and subject to change.]

The CP Broadcast Property ID field contains a foreign key from the Broadcast Property table. This field's values are as follows:

Key: Primary
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Not applicable
Required: No

## CP Channel ID

[This is preliminary documentation and subject to change.]

The CP Channel ID field contains a foreign key from the Channel table. This field's values are as follows:

Key: Primary
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## Channel Stream Table

[This is preliminary documentation and subject to change.]

The Channel Stream table in the Guide database lists the data streams available on a specific channel. The Channel Stream table contains the following fields.

| **Field** | **Description** |
|---|---|
| CSR Channel ID | Unique identifier for this channel. |
| CSR Name | Name of the stream. |
| CSR Stream Type ID | Unique identifier for this stream type. |
| CSR SubChannel | Subchannel relative to the channel specified by CSR Channel ID. |

## CSR Channel ID

[This is preliminary documentation and subject to change.]

The CSR Channel ID field contains the unique identifier for this channel. This field's values are as follows:

Key: Primary, foreign
Index: No

Data type: Number
Field size: Long integer
Default value: Zero
Required: Yes

## CSR Name

[This is preliminary documentation and subject to change.]

The CSR Name field contains the user-oriented label used to identify this stream in this configuration. The CSR Name value overrides the description in the Stream Type table. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Memo
Field size: Not applicable
Default value: Zero-length string
Required: No

## CSR Stream Type ID

[This is preliminary documentation and subject to change.]

The CSR Stream Type ID field contains the unique identifier for this stream type. This field's values are as follows:

Key: Primary, foreign
Index: No
Data type: Number
Field size: Long integer
Default value: Zero
Required: Yes

## CSR SubChannel

[This is preliminary documentation and subject to change.]

The CSR SubChannel field contains the subchannel number used to identify this stream relative to the channel. This field's values are as follows:

Key: Primary
Index: No
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## Enhancement Table

[This is preliminary documentation and subject to change.]

The Enhancement table in the Guide database contains information used by TV Viewer to link shows to their enhancements. The Enhancement table contains the following fields.

| Field | Description |
|---|---|
| EN Address | Internet Protocol (IP) address of the enhancement stream. |
| EN Enhancement ID | Identifying name for this enhancement. |
| EN Expire Date | Expiration date extracted from a show reference. This date can be used in a deletion query to update this table. |
| EN Mapping ID | Identifier used by TV Viewer for efficient display of enhancements. |
| EN Preload URL | Home or start page for this enhancement. |
| EN Show Reference | Value that identifies one or more episodes to be enhanced. |
| EN Title | Title used to display the enhancement in the user interface. |

### EN Address

[This is preliminary documentation and subject to change.]

The EN address field is the IP address of the enhancement data. This field's values are as follows:

Key: Primary
Index: No
Data type: Memo
Field size: Not applicable
Default value: Zero-length string
Required: No

## EN Enhancement ID

[This is preliminary documentation and subject to change.]

The EN Enhancement ID field contains a unique identifier for the enhancement. This field's values are as follows:

Key: Primary
Index: Yes (no duplicates)
Data type: Text
Field size: 38 characters
Default value: Zero-length string
Required: Yes

## EN Expire Date

[This is preliminary documentation and subject to change.]

The EN Expire Date field is used to determine when to remove this enhancement from the database. This field's values are as follows:

Key: Primary
Index: No
Data type: Date
Field size: Long integer
Default value: NULL date
Required: No

## EN Mapping ID

[This is preliminary documentation and subject to change.]

The EN Mapping ID field is used by TV Viewer for the efficient display of enhancements. This field is automatically generated when the enhancement record is created. This field's values are as follows:

Key: Primary
Index: Yes
Data type: AutoNumber
Field size: Long integer
Default value: Automatically incremented
Required: Yes

## EN Preload URL

[This is preliminary documentation and subject to change.]

The EN Preload URL field is the Uniform Resource Locator (URL) of the data to download for this enhancement. Preloading this data enables the enhancement to include more data than the available bandwidth otherwise allows. This field's values are as follows:

Key: Primary
Index: No
Data type: Memo
Field size: Not applicable
Default value: Zero-length string
Required: No

## EN Show Reference

[This is preliminary documentation and subject to change.]

The EN Show Reference field is the identifier of the show that uses this enhancement. This field's values are as follows:

Key: Primary
Index: Yes (duplicates okay)
Data type: Memo
Field size: Not applicable

Default value: Zero-length string
Required: No

## EN Title

[This is preliminary documentation and subject to change.]

The EN Title field is the title of the enhancement. This field's values are as follows:

Key: Primary
Index: Yes
Data type: String
Field size: 255 characters
Default value: Zero-length string
Required: No

# Episode Table

[This is preliminary documentation and subject to change.]

The Episode table in the Guide database describes each television program episode. The Episode table contains the following fields.

| **Field** | **Description** |
|---|---|
| E Abbreviation | Bit field that indicates how to abbreviate episode titles for smaller display areas. |
| E Description | Extended description of this episode. |
| E Display Mask | Reserved for future use. |
| E Episode ID | Internally generated unique identifier for this episode. |
| E Last Update | Last time this row in the table was updated. |
| E Rating ID | Foreign key from the Rating table. |
| E Theme ID | Foreign key from the Theme table. |
| E Title | Program title. |

## E Abbreviation

[This is preliminary documentation and subject to change.]

The E Abbreviation field contains a bit field that indicates how to abbreviate episode titles for smaller display areas. For example, "How to Succeed in Business Without Even Trying" appears as "Succeed Business Without Even Trying" when this field has a binary value of 0010111. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## E Description

[This is preliminary documentation and subject to change.]

The E Description field contains an extended description of this episode (for example, "Mail order diplomas investigated"). This field's values are as follows:

Key: Not applicable
Index: No
Data type: Text
Field size: 255 characters
Default value: Zero-length string
Required: No

## E Display Mask

[This is preliminary documentation and subject to change.]

Reserved for future use.

## E Enhancement Mapping ID

[This is preliminary documentation and subject to change.]

The E Attachment field contains a URL or application associated with this episode. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Memo
Field size: Not applicable
Default value: Zero-length string
Required: No

## E Episode ID

[This is preliminary documentation and subject to change.]

The E Episode ID field provides an internally generated unique identifier for this episode. This field's values are as follows:

Key: Primary
Index: Yes (no duplicates)
Data type: AutoNumber
Field size: Long integer
Default value: Automatically incremented
Required: Yes

## E Last Update

[This is preliminary documentation and subject to change.]

The E Last Update field contains the last time this row in the table was updated. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Date/time
Field size: Not applicable
Default value: Not applicable

Required: No

## E Rating ID

[This is preliminary documentation and subject to change.]

The E Rating ID field contains a foreign key from the Rating table. This field's values are as follows:

Key: Foreign
Index: No
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## E Theme ID

[This is preliminary documentation and subject to change.]

The E Theme ID field contains a foreign key from the Theme table. This field's values are as follows:

Key: Foreign
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## E Title

[This is preliminary documentation and subject to change.]

The E Title field contains the program title (for example, Frontline). This field's values are as follows:

Key: Not applicable
Index: Yes (duplicates okay)
Data type: Text
Field size: 255 characters

Default value: Not applicable
Required: Yes

# Episode Property Table

[This is preliminary documentation and subject to change.]

The Episode Property table in the Guide database specifies properties associated with individual program episodes. The Episode Property table contains the following fields.

| Field | Description |
|---|---|
| Broadcast Property ID | Foreign key from the Broadcast Property table. |
| Episode ID | Foreign key from the Episode table. |

## Broadcast Property ID

[This is preliminary documentation and subject to change.]

The Broadcast Property ID field contains a foreign key from the Broadcast Property table. This field's values are as follows:

Key: Foreign
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Not applicable
Required: Yes

## Episode ID

[This is preliminary documentation and subject to change.]

The Episode ID field contains a foreign key from the Episode table. This field's values are as follows:

Key: Foreign
Index: Yes (duplicates okay)

Data type: Number
Field size: Long integer
Default value: Zero
Required: Yes

## Genre Table

[This is preliminary documentation and subject to change.]

The Genre table in the Guide database provides names for different categories of program episodes. The Genre table contains the following fields.

| **Field** | **Description** |
|---|---|
| G Genre ID | Unique identifier for this record. |
| G Name | Name of this genre (for example, movies, television series, and so on). |
| G Tuning Space | Numeric identifier for the data source that defines this genre. |

## G Genre ID

[This is preliminary documentation and subject to change.]

The G Genre ID field contains the unique identifier for this genre. This field's values are as follows:

Key: Primary
Index: Yes (no duplicates)
Data type: AutoNumber
Field size: Long integer
Default value: Automatically incremented
Required: Yes

## G Name

[This is preliminary documentation and subject to change.]

The G Name field contains the name of this genre (for example, movies, television series, and so on). This field's values are as follows:

Key: Not applicable
Index: Yes (duplicates okay)
Data type: Text
Field size: 50 characters
Default value: Not applicable
Required: No

### G Tuning Space

[This is preliminary documentation and subject to change.]

The G Tuning Space field contains a numeric identifier for the data source that defines this genre. This value identifies which loader handles the source data. This field's values are as follows:

Key: Not applicable
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## Network Table

[This is preliminary documentation and subject to change.]

The Network table in the Guide database defines information about various broadcast networks. The Network table contains the following fields.

| **Field** | **Description** |
|---|---|
| N LogoMoniker | Name of the image file that contains the network logo. |
| N Name | Name of the network. |
| N Network ID | Automatically generated unique identifier for the network. |

## N LogoMoniker

[This is preliminary documentation and subject to change.]

The N LogoMoniker field provides the name of the bitmap file containing the logo for this network. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Text
Field size: 255 characters
Default value: Not applicable
Required: No

## N Name

[This is preliminary documentation and subject to change.]

The N Name field provides the name of this network. This field's values are as follows:

Key: Not applicable
Index: Yes (no duplicates)
Data type: Text
Field size: 50 characters
Default value: Zero-length string
Required: No

## N Network ID

[This is preliminary documentation and subject to change.]

The N Network ID field provides an internally generated unique identifier for this network. This field's values are as follows:

Key: Primary
Index: Yes (no duplicates)
Data type: AutoNumber
Field size: Long integer
Default value: Automatically incremented

Required: Yes

## Rating Table

[This is preliminary documentation and subject to change.]

The Rating table in the Guide database defines rating names for individual ratings used in various rating systems. Any time an application changes an entry in the Rating table, that application must update the .rat file with new descriptions. Each record in the Rating table is roughly equivalent to a label entry in a Platform for Internet Content Selection (PICS) rating file. The Rating table contains the following fields.

| **Field** | **Description** |
|---|---|
| R Description | Name of this rating (for example, PG-13) in the rating system. |
| R Pictogram Moniker | File name of the bitmap associated with this rating. |
| R Rating | Numerical value used to rank this rating in the rating system. |
| R Rating ID | Unique identifier for this record. |
| R Rating System ID | Identifier for the rating system that uses this rating entry. |

### R Description

[This is preliminary documentation and subject to change.]

The R Description field contains the name of this rating (for example, PG-13) in the rating system. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Text
Field size: 50 characters
Default value: Not applicable
Required: No

## R Pictogram Moniker

[This is preliminary documentation and subject to change.]

The R Pictogram Moniker field contains the file name of the bitmap associated with this rating. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Text
Field size: 255 characters
Default value: Zero-length string
Required: No

## R Rating

[This is preliminary documentation and subject to change.]

The R Rating field contains the numerical value used to rank this rating in the rating system. This field's values are as follows:

Key: Not applicable
Index: Yes (duplicates okay)
Data type: Number
Field size: Single
Default value: Zero
Required: No

## R Rating ID

[This is preliminary documentation and subject to change.]

The R Rating ID field contains a unique identifier for this rating. This field's values are as follows:

Key: Primary
Index: Yes (no duplicates)
Data type: AutoNumber
Field size: Long integer
Default value: Automatically incremented
Required: Yes

## R Rating System ID

[This is preliminary documentation and subject to change.]

The R Rating System ID field contains an identifier for the rating system using this rating entry. This field's values are as follows:

Key: Foreign
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

# Rating System Table

[This is preliminary documentation and subject to change.]

The Rating System table in the Guide database provides names for MPAA-style ratings. The Rating System table contains the following fields.

| **Field** | **Description** |
|---|---|
| RS Description | Description of the rating system. |
| RS Name | Name of the rating system. |
| RS Rating System ID | Unique identifier for this record. |
| RS Tuning Space | Data source defining this rating system. |

## RS Description

[This is preliminary documentation and subject to change.]

The RS Description field contains the description of this rating system. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Text
Field size: 128 characters
Default value: No
Required: No

## RS Name

[This is preliminary documentation and subject to change.]

The RS Name field contains the name of this rating system (for example, MPAA). This field's values are as follows:

Key: Not applicable
Index: Yes (no duplicates)
Data type: Text
Field size: 50 characters
Default value: Not applicable
Required: No

## RS Rating System ID

[This is preliminary documentation and subject to change.]

The RS Rating System ID field contains the unique identifier for this rating system. This field's values are as follows:

Key: Primary
Index: Yes (no duplicates)
Data type: AutoNumber
Field size: Not applicable
Default value: Automatically incremented
Required: Yes

## RS Tuning Space

[This is preliminary documentation and subject to change.]

The RS Tuning Space field contains the data source defining this rating system. This value identifies which loader handles the source data. This field's values are as follows:

Key: Not applicable
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## Station Table

[This is preliminary documentation and subject to change.]

The Station table in the Guide database defines individual broadcast content providers. The Station table contains the following fields.

| Field | Description |
|---|---|
| S Call Letters | Call letters of this station (HBOW, for example). |
| S Description | Extended description of the programming provided by this station. |
| S Logo | Name of the file that contains the logo bitmap for this station. |
| S Name | Name of this station (Home Box Office West, for example). |
| S Network ID | Foreign key from the Network table. |
| S Station ID | Internally generated unique identifier for this station. |

## S Call Letters

[This is preliminary documentation and subject to change.]

The S Call Letters field provides the call letters of this station (HBOW, for example). This field's values are as follows:

Key: Not applicable
Index: Yes (no duplicates)
Data type: Text
Field size: 5 characters
Default value: Not applicable
Required: Yes

## S Description

[This is preliminary documentation and subject to change.]

The S Description field provides an extended description of the programming provided by this station. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Memo
Field size: Not applicable
Default value: Zero-length string
Required: No

## S Logo

[This is preliminary documentation and subject to change.]

The S Logo field provides the name of the file containing the logo bitmap for this station. The file suffix is not included in this string. For example, the S Logo value for the Program Guide record is Msepg, not Msepg.bmp. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Text
Field size: 255 characters
Default value: Zero-length string
Required: No

## S Name

[This is preliminary documentation and subject to change.]

The S Name field provides the name of this station (Home Box Office West, for example). This field's values are as follows:

Key: Not applicable
Index: No
Data type: Text
Field size: 50 characters
Default value: Zero-length string
Required: No

## S Network ID

[This is preliminary documentation and subject to change.]

The S Network ID field is a foreign key from the Network table. This field's values are as follows:

Key: Foreign
Index: No
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## S Station ID

[This is preliminary documentation and subject to change.]

The S Station ID field provides an internally generated unique identifier for this station. This field's values are as follows:

Key: Primary
Index: Yes (no duplicates)
Data type: AutoNumber
Field size: Long integer
Default value: Automatically incremented
Required: Yes

## Stream Type Table

[This is preliminary documentation and subject to change.]

The Stream Type table in the Guide database gives names for all data streams available on a specific channel. The Stream Type table contains the following fields.

| Field | Description |
|---|---|
| SR Category | Value that specifies the category of the stream. |
| SR Description | Name of this stream (for example, video, data, and so on). |
| SR Locale ID | Locale identifier of the stream. |
| SR Stream Type ID | Unique identifier for this stream type. |
| SR Tuning Space | Value that identifies the source that defines this stream. |
| SR Value | Integer passed to the driver when requesting a specific video stream. |

### SR Category

[This is preliminary documentation and subject to change.]

The SR Category field specifies the category for the stream. This field's values are as follows:

Key: Not applicable
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: Yes

### SR Description

[This is preliminary documentation and subject to change.]

The SR Description field contains the name of this stream (for example, video, data, and so on). This field's values are as follows:

Key: Not applicable
Index: No
Data type: Memo
Field size: Not applicable
Default value: Zero-length string
Required: No

## SR Locale ID

[This is preliminary documentation and subject to change.]

The SR Locale ID field specifies the locale identifier for the stream. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## SR Stream Type ID

[This is preliminary documentation and subject to change.]

The SR Stream Type ID field contains the unique identifier for this stream type. This field's values are as follows:

Key: Primary
Index: Yes (no duplicates)
Data type: AutoNumber
Field size: Long integer
Default value: Automatically incremented
Required: Yes

## SR Tuning Space

[This is preliminary documentation and subject to change.]

The SR Tuning Space field identifies the source that defines this stream. This field's values are as follows:

Key: Not applicable
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## SR Value

[This is preliminary documentation and subject to change.]

The SR Value field contains the integer passed to the driver when requesting a specific video stream. This field's values are as follows:

Key: Not applicable
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

# Sub-Genre Table

[This is preliminary documentation and subject to change.]

The Sub-Genre table in the Guide database provides names for subcategories within a genre. The Sub-Genre table contains the following fields.

| **Field** | **Description** |
|---|---|
| SG Name | Name of this subgenre (for example, science fiction). |
| SG Sub Genre ID | Unique identifier for this record. |

### SG Name

[This is preliminary documentation and subject to change.]

The SG Name field contains the name of this subgenre (for example, science fiction). This field's values are as follows:

Key: Not applicable
Index: Yes (duplicates okay)
Data type: Text
Field size: 50 characters
Default value: Not applicable
Required: No

### SG Sub Genre ID

[This is preliminary documentation and subject to change.]

The SG Sub Genre ID field contains the unique identifier for this subgenre. This field's values are as follows:

Key: Not applicable
Index: Yes (no duplicates)
Data type: AutoNumber
Field size: Long integer
Default value: Automatically incremented
Required: Yes

## Theme Table

[This is preliminary documentation and subject to change.]

The Theme table in the Guide database defines information about themes. A theme allows an episode to be associated with multiple genre/subgenre pairs, because genre/subgenre pairs can share the same T Theme ID field.

For example, the motion picture *Aliens* might have a T Theme ID of 2, placed in the E Theme ID field of the Episode table. In the Theme table, the genre/subgenre pair Science Fiction/Movie and the genre/subgenre pair Movie/Action can both share that same T Theme ID of 2.

The Theme table contains the following fields.

| Field | Description |
|---|---|
| T Genre ID | Foreign key from the Genre table. |
| T Sub Genre ID | Foreign key from the Sub-Genre table. |
| T Theme ID | Internally generated unique identifier for this record. |

## T Genre ID

[This is preliminary documentation and subject to change.]

The T Genre ID field contains a foreign key from the Genre table. This field's values are as follows:

Key: Primary
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: Yes

## T Sub Genre ID

[This is preliminary documentation and subject to change.]

The T Sub Genre ID field contains a foreign key from the Sub Genre table. This field's values are as follows:

Key: Primary
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: Yes

## T Theme ID

[This is preliminary documentation and subject to change.]

The T Theme ID field contains the internally generated unique identifier for this theme. This field's values are as follows:

Key: Primary
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: Yes

## Theme ID Mapping Table

[This is preliminary documentation and subject to change.]

The Theme ID Mapping table in the Guide database maps a genre, subgenre, or genre/subgenre pair to a corresponding theme identifier. The Theme ID Mapping table contains the following fields.

| **Field** | **Description** |
|---|---|
| TM Genre Identifier | Identifier of the genre to map. Either this field or the TM Genre Name field can be used to specify the genre. |
| TM Genre Name | Name of the genre to map. Either this field or the TM Genre Identifier field can be used to specify the genre. |
| TM SubGenre Identifier | Identifier of the subgenre to map. Either this field or the TM SubGenre Name field can be used to specify the subgenre. |
| TM SubGenre Name | Name of the subgenre to map. Either this field or the TM SubGenre Identifier field can be used to specify the subgenre. |
| TM Theme ID | Theme identifier. |
| TM Tuning Space | Tuning space that this mapping applies to. |

## TM Genre Identifier

[This is preliminary documentation and subject to change.]

The TM Genre Identifier field specifies the genre to map. Either this field or TM Genre Name can be use to specify the genre. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## TM Genre Name

[This is preliminary documentation and subject to change.]

The TM Genre Name field specifies the genre to map. Either this field or TM Genre Identifier can be use to specify the genre. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Text
Field size: 255 characters
Default value: Not applicable
Required: No

## TM SubGenre Identifier

[This is preliminary documentation and subject to change.]

The TM SubGenre Identifier field specifies the subgenre to map. Either this field or TM SubGenre Name can be use to specify the subgenre. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Number

Field size: Long integer
Default value: Zero
Required: No

## TM SubGenre Name

[This is preliminary documentation and subject to change.]

The TM SubGenre Name field specifies the subgenre to map. Either this field or TM SubGenre Identifier can be use to specify the subgenre. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Text
Field size: 255 characters
Default value: Not applicable
Required: No

## TM Theme ID

[This is preliminary documentation and subject to change.]

The TM Theme ID field specifies the theme identifier to which the genre, subgenre, or genre/subgenre pair should be mapped to. This field's values are as follows:

Key: Not applicable
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Not applicable
Required: No

## TM Tuning Space

[This is preliminary documentation and subject to change.]

The TM Tuning Space field specifies the tuning space for which this mapping applies. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Number
Field size: Long integer
Default value: Not applicable
Required: No

# Time Slot Table

[This is preliminary documentation and subject to change.]

The Time Slot table in the Guide database defines time slot and channel information for each program episode. The Time Slot table contains the following fields.

| Field | Description |
|---|---|
| TS Alternate Audio Exists | Value that indicates whether an alternate audio stream is available. |
| TS Alternate Data Exists | Value that indicates whether an alternate data stream is available. |
| TS Channel ID | Foreign key from the Channel table. |
| TS Closed Caption | Value that specifies whether the broadcast has closed captions. |
| TS End Time | Ending time for this time slot. |
| TS Episode ID | Foreign key from the Episode table. |
| TS Last Update | Last time this row in the table was updated. |
| TS Length | Length of this time slot, in minutes. |
| TS Other Properties Exist | Value that indicates whether additional properties exist for the broadcast. |
| TS Pay Per View | Reserved for future use. |
| TS Payment Address | Reserved for future use. |
| TS Payment Token | Reserved for future use. |
| TS Rerun | Value that specifies whether the broadcast is a rerun. |
| TS Start Time | Beginning time for this time slot. |
| TS Stereo | Value that specifies whether the broadcast is in stereo. |
| TS Tape Inhibited | Reserved for future use. |

| | |
|---|---|
| TS Time Slot ID | Unique identifier for this time slot record. |

## TS Alternate Audio Exists

[This is preliminary documentation and subject to change.]

The TS Alternate Audio Exists field indicates whether an alternate audio stream is available. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Yes/no
Field size: Not applicable
Default value: Zero
Required: No

## TS Alternate Data Exists

[This is preliminary documentation and subject to change.]

The TS Alternate Data Exists field indicates whether an alternate data stream is available. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Yes/no
Field size: Not applicable
Default value: Zero
Required: No

## TS Channel ID

[This is preliminary documentation and subject to change.]

The TS Channel ID field contains a foreign key from the Channel table. This field's values are as follows:

Key: Primary, foreign
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## TS Closed Caption

[This is preliminary documentation and subject to change.]

The TS Closed Caption field specifies whether the broadcast has closed captions. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Yes/no
Field size: Not applicable
Default value: Zero
Required: No

## TS End Time

[This is preliminary documentation and subject to change.]

The TS End Time field contains the ending time for this time slot. This field's values are as follows:

Key: Primary
Index: Yes (duplicates okay)
Data type: Date/time
Field size: Not applicable
Default value: Not applicable
Required: Yes

## TS Episode ID

[This is preliminary documentation and subject to change.]

The TS Episode ID field contains a foreign key from the Episode table. This field's values are as follows:

Key: Primary
Index: Yes (duplicates okay)
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## TS Last Update

[This is preliminary documentation and subject to change.]

The TS Last Update field contains the last time this row in the table was updated. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Date/time
Field size: Not applicable
Default value: Not applicable
Required: Yes

## TS Length

[This is preliminary documentation and subject to change.]

The TS Length field contains the length of this time slot in minutes. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Number
Field size: Long integer
Default value: Zero
Required: No

## TS Other Properties Exist

[This is preliminary documentation and subject to change.]

The TS Other Properties Exist field indicates whether additional properties exist for the broadcast. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Yes/no
Field size: Not applicable
Default value: Zero
Required: No

## TS Pay Per View

[This is preliminary documentation and subject to change.]

Reserved for future use.

## TS Payment Address

[This is preliminary documentation and subject to change.]

Reserved for future use.

## TS Payment Token

[This is preliminary documentation and subject to change.]

Reserved for future use.

## TS Rerun

[This is preliminary documentation and subject to change.]

The TS Rerun field specifies whether the broadcast is a rerun. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Yes/no
Field size: Not applicable
Default value: Zero
Required: No

## TS Start Time

[This is preliminary documentation and subject to change.]

The TS Start Time field contains the beginning time for this time slot. This field's values are as follows:

Key: Primary
Index: Yes (duplicates okay)
Data type: Date/time
Field size: Not applicable
Default value: Not applicable
Required: Yes

## TS Stereo

[This is preliminary documentation and subject to change.]

The TS Stereo field specifies whether the broadcast is in stereo. This field's values are as follows:

Key: Not applicable
Index: No
Data type: Yes/no
Field size: Not applicable
Default value: Zero
Required: No

## TS Tape Inhibited

[This is preliminary documentation and subject to change.]

Reserved for future use.

## TS Time Slot ID

[This is preliminary documentation and subject to change.]

The TS Time Slot ID field contains a unique identifier for this time slot record. This field's values are as follows:

Key: Not applicable
Index: Yes (no duplicates)
Data type: AutoNumber
Field size: Long integer
Default value: Automatically incremented
Required: Yes

# Program Guide Registry Entries

[This is preliminary documentation and subject to change.]

*Guide database* loaders, to do their job properly, depend on information stored in the computer's registry. The registry entries that provide this information are found in the following sections:

- Tuning Spaces
- Loaders

# Video Control

[This is preliminary documentation and subject to change.]

In Broadcast Architecture, you can use the Microsoft® ActiveX™ control for video (the Video control, Vid.ocx) to manage and display multimedia streams in Web pages and applications. Although the Video control can also be used to display video in enhancement layout pages, it is usually more convenient to use the Enhancement control. For more information, see Which Video Control Should I Use?

The following topics describe the Video control, explain how to use it in an application, and outline reference information for the object classes implemented by the Video control:

- About the Video Control
- Using the Video Control
- Video Control Reference

# About the Video Control

[This is preliminary documentation and subject to change.]

Your application can use the Microsoft® ActiveX™ control for video (the Video control, Vid.ocx) to manage audio and video streams. This control also enables your application to interact with the *Video Access server*, Vidsvr.exe, to manage user purchase of broadcast items, such as pay-per-view shows.

The input stream that the Video control displays can be a live stream, such as an analog television broadcast received through a television *tuner* card.

For more information, see the following topics:

- Available Devices
- Event Notification
- Device Contention
- Reserved Classes, Events, and Properties

## Available Devices

[This is preliminary documentation and subject to change.]

When a *broadcast client* system starts, the Video Access server creates a list of the devices available on the client. Using this list, your application can programmatically query and control individual devices.

This list is available to the Video control through the **BPCDevices** collection. Each **BPCDeviceBase** object in this collection represents a particular device on the client. The properties and methods of **BPCDeviceBase** can programmatically query and control these devices.

The Video Access server builds this list of available multimedia devices using the Plug and Play enumeration in the Microsoft® DirectShow™ application programming interface (API). You can then set the Video control input device to one of the enumerated devices.

## Event Notification

[This is preliminary documentation and subject to change.]

The Video control sends events relating to itself such as **BPCVid.**DblClick, which is raised when a user double-clicks inside the control, to your application. The Video control uses events to implement device contention resolution. It uses the **BPCVid.**GotControl and **BPCVid.**LostControl methods to notifiy applications when they have or lose control of a particular device. For more information, see Device Contention.

To receive event notifications from the Video control, you must implement event-handler methods in your application. These methods are called by the Video control when an event occurs.

## Device Contention

[This is preliminary documentation and subject to change.]

The Video control uses a cooperative mechanism to prevent device conflicts. This prevents multiple instances of the Video control from attempting to control a particular device at the same time, which would cause unpredictable results.

The **BPCVid.**GotControl and **BPCVid.**LostControl event notifications are used to inform an application that it has control or must relinquish control of a particular device.

For example, if your application needs to use a particular device it must first request control of the device by setting the input or output property of the Video control equal to the device. Your application should then wait until it receives a GotControl notification before it uses the device. If the application attempts to access the device before it has control, unexpected behavior can occur.

Similarly, if your application receives a LostControl notification, it should immediately stop using the device in question until it can regain control. Otherwise, unexpected behavior occurs when multiple applications access the device.

Thus, in order to handle device contention, your application must implement GotControl and LostControl event handlers.

The device contention protocol also implements priority levels for application access to devices. These priority levels are used when two applications are in contention for a device. In this case, control is given to the application with higher priority. If both applications have equal priority, the application that currently has control of the device retains control. You can set the priority for an instance of the Video control using the **BPCVid.Priority** property.

## Reserved Classes, Events, and Properties

[This is preliminary documentation and subject to change.]

The Video control contains several classes, events, and properties that are not supported and are reserved for future use.

The following object classes are reserved:

- **BPCPurchase**
- **BPCHistoryItemsCollection**
- **BPCMessage**
- **BPCEmailMessage**
- **BPCEmailMessageCollection**

The following properties and methods of **BPCDeviceBase** are reserved:

- **Status**
- **Filename**
- **HasFilename**
- **BuyItem**
- **CancelItem**
- **CurrentState**
- **EmailMessages**
- **HandleCardChaining**
- **HasCA**
- **HistoryItems**
- **ItemDetails**
- **ProviderEPGMask**
- **ProviderRating**
- **ProviderStatus**

- **ResetProviderSystem**
- **Run**
- **Stop**
- **Pause**

The following methods and events of **BPCVid** and **BPCDevices** are reserved:

- **Login**
- **Logout**
- **Run**
- **Stop**
- **Pause**
- BlackedOut
- CAFail
- CAFault
- CannotPurchase
- CardInvalid
- CardMissing
- CASuccess
- ColdStart
- CopyCard
- CostExceeded
- EPGFilterChanged
- HandlePurchaseOffer
- NewEmail
- NoSubscriber
- NotReady
- OSDRequest
- PasswordCleared
- RatingExceeded
- Ready
- Retry
- RevokeEvent
- SignalLost
- StateChange
- TapingControlChanged
- TuningChanged
- WrongCard

## Using the Video Control

[This is preliminary documentation and subject to change.]

The Microsoft® ActiveX™ control for video (the Video control) is a dual-interface ActiveX control. This functionality means you can use the control from a variety of programming environments, such as

World Wide Web pages, applications built in the Microsoft® Visual Basic® programming system, Java applets and applications, and C++ applications.

For details on working with the Video control to display video streams, manage devices, and interact with the Video Access server, see:

- Setting an Input Device
- Setting an Output Device
- Setting a Input Channel
- Suspending the Video Server

For additional information on the tasks involved in using the Video control, see Displaying Video. In addition, Broadcast Architecture material includes sample applications that demonstrate use of the Video control. To locate these samples, see Broadcast Architecture Sample Applications.

## Setting an Input Device

[This is preliminary documentation and subject to change.]

In order to display a video stream, your application must first set an input device or source for that stream. Your application does this by setting the **BPCVid.Input** property of the Video control equal to one of the available devices in the **BPCDevices** collection.

There are many different types of input devices. Your application must ensure that any device it selects as an input device supports the operations that it is trying to perform. To do so, your application can use properties of the form **Has***Xxxx*, which indicate whether the current device supports the property *Xxxx*.

For example, the **BPCDeviceBase.HasChannel** indicates whether the device supports channels and can be tuned by calling **BPCDeviceBase.Channel**. The **HasChannel** property of a television *tuner* card is True.

If there are multiple tuners available on a broadcast client, your application can use the **BPCDeviceBase.ChannelAvailable** method to determine whether a device can tune to the specified channel.

Once your application has selected a device that supports the intended video stream from **BPCDevices**, your application can set that device as the input device. This process is illustrated in the following example:

```
Dim Device

'Search the collection for a device that
'supports channels. When one is found, set the
'that device as the input device.

For Each Device In MyVid.Devices
```

```
  If Device.HasChannel Then
    MyVid.Input = Device
    Exit For
  End If
Next
```

## Setting an Output Device

[This is preliminary documentation and subject to change.]

You can also use the Video control to send video streams to an output device. For example, your application can record a television broadcast by setting a tuner card as the input device and a VCR as the output device.

**Note** Broadcast Architecture does not support functionality to enable infrared (IR) frequency transmission, such as might be used to emulate a remote control signal and control consumer electronics. Thus, although you can set a VCR or other output device to receive a video stream, the user still has to turn on the VCR and start recording manually.

Setting an output device is similar to setting an input device, but instead of using the **BPCVid.Input** property your application uses the **BPCVid.Output** property.

The following example searches the Devices collection for a device that supports output. When one is found, the example sets that device as the output device.

```
Dim Device

For Each Device In MyVid.Devices
  If Device.IsOutput Then
    MyVid.Output = Device
    Exit For
  End If
Next
```

## Setting a Input Channel

[This is preliminary documentation and subject to change.]

Once you have set an input device for the Video control that supports channels, as described in Setting an Input Device, you can use the **BPCDeviceBase.Channel** property to set the device to the appropriate channel.

However, if multiple *tuning spaces* are defined on a broadcast client, as happens when there are multiple tuner cards installed, a particular device might not be able to tune to the specified channel. You can check whether a device supports a channel by using the **BPCDeviceBase.ChannelAvailable** method.

The following example loops through the **BPCDevices** collection, searching for a device that supports the specified channel. If such a device is found, the script sets that device as the input device and sets the device's **Channel** property.

```
Dim Device
Dim MyChannel = 12

For Each Device In MyVid.Devices
  If MyVid.Input.HasChannel Then
    If Device.ChannelAvailable(MyChannel)
      MyVid.Input = Device
      MyVid.Input.Channel = MyChannel
      Exit For
    End If
  End If
Next
```

## Suspending the Video Server

[This is preliminary documentation and subject to change.]

You can use the **BPCSuspend** object to suspend background data capture and cause the video server to release its devices for use by other applications.

To do this you create an instance of **BPCSuspend** and call its **DeviceRelease** method. If the video server successfully releases all devices the method returns a valid **BPCSuspended** object. Otherwise, it returns **Nothing**. If **Nothing** is returned, it means that the video server was unable to release some devices because they are currently being used by client video applications.

Your application should handle the case where **DeviceRelease** returns **Nothing** as it would a device busy or device open type of failure. Your application can wait and try again, or signal the user to shut down video applications.

When your application is done using the devices, it should destroy the **BPCSuspended** object. This notifies the video server that it can resume using the devices and return to background data capture.

```
Dim susp As BPCSuspend
Dim suspended As BPCSuspended
Set susp = New BPCSuspend

'Attempt to suspend the video server
susp.DeviceRelease( 0, suspended )

'Test whether the video server was suspended
If suspended = Nothing Then
```

```
  'Handle the case where devices cannot be released
Else
  'Use the devices
End If

'Release suspended to enable the video server to use the devices
Set suspended = Nothing
```

**Note** If you are developing applications using C/C++, you can use the wrapper class implemented in Bpcsusp.h to provide suspend functionality.

## Video Control Reference

[This is preliminary documentation and subject to change.]

The following sections document the programmatic interfaces of the Microsoft® ActiveX™ control for video (the Video control, Vid.ocx) and the Video Access server, Vidsvr.exe. These reference topics assume use by developers in the Microsoft® Visual Basic® programming system and Microsoft® Visual Basic® Scripting Edition (VBScript) scripting language. Therefore, only Visual Basic language syntax is shown.

The following objects are provided by the **MSBPCVideo** object library, Vid.ocx.

| **Object** | **Description** |
|---|---|
| **BPCVid** | The Video control. |
| **BPCDevices** | A collection of all available devices on the user's computer. |
| **BPCDeviceBase** | A device. |
| **BPCSuspend** | Object that can be used to suspend the video server and get it to release all devices. |
| **BPCSuspended** | Object that indicates whether the video server is suspended. |

## Object References

[This is preliminary documentation and subject to change.]

In the object reference topics, the syntax placeholder *object* refers to an object expression that evaluates to one of the classes specified.

The **Input** and **Output** properties of the Video control are each assigned **DeviceBase** objects when setting the properties' values. You can specify the properties of a **DeviceBase** object in either property with the following syntax:

*Vid1*.**Devices**(*index*).*property*

where *Vid1, index,* and *property* are placeholders respectively for the:

- Specific Video control.
- Index to the specific **DeviceBase** object in the **Devices** collection.
- Property you want to use.

You can also specify **DeviceBase** object properties after you assign them to the **Input** or **Output** properties of the Video control. These are also valid object expressions that reference a property of a **DeviceBase** object:

*Vid1*.**Input**.*property*
*Vid1*.**Output**.*property*

where *Vid1* and *property* are the respective placeholders for the specific Video control you are using and the property to set of the associated **DeviceBase** object.

## Visual Basic Extender Object

[This is preliminary documentation and subject to change.]

Some properties of a control are provided by the container rather than the control; these are extender properties. The control still needs information on, and sometimes needs to change the value of, these properties. To provide this functionality, Visual Basic provides an **Extender** object to access these properties. Some extender properties are standard; others are specific to certain containers.

The **Extender** object has several standard properties that affect the Video control. Visual Basic provides additional extender methods, properties, and events that may not be available in other containers. The following tables list and describe these **Extender** object and Visual Basic–specific properties, methods, and events. The properties are as follows.

| **Property** | **Description** |
|---|---|
| **Container** | Object that represents the visual container of the control. |
| **DragIcon** | Picture that specifies the icon to use when the control is dragged. |

| | |
|---|---|
| **DragMode** | **Integer** that specifies if the control automatically drags, or if the application using the control must call the **Drag** method. |
| **Enabled** | Boolean value that specifies whether the control is enabled. |
| **Height** | **Integer** that specifies the height of the control in the container's scale units. |
| **HelpContextID** | **Integer** that specifies the context identifier to use when the F1 key is pressed and the control has the focus. |
| **Index** | **Integer** that specifies the position in a control array this instance of the control occupies. |
| **Left** | **Integer** that specifies the position of the left edge of the control relative to the left edge of the container, specified in the container's scale units. |
| **Name** | **String** that contains the user-defined name of the control. |
| **Parent** | Object that represents the container of the control, such as a form in Visual Basic. |
| **TabIndex** | **Integer** that specifies the position of the control in the tab order of the controls in the container. |
| **TabStop** | Boolean value that specifies if the tab stops on the control. |
| **Tag** | **String** that contains a user-defined value. |
| **ToolTipText** | **String** that contains the text to be displayed when the cursor hovers above the control for more than a second. |
| **Top** | **Integer** that specifies the position of the upper edge of the control relative to the upper edge of the container, specified in the container's scale units. |
| **Visible** | Boolean value that specifies whether the control is visible. |
| **WhatThisHelpID** | Context identifier to use when the "What's This" pop-up is used to provide information on the control. |
| **Width** | Width of the control in the container's scale units. |

The **Extender** object methods and Visual Basic–specific extender methods are listed following.

| **Method** | **Description** |
|---|---|
| **Drag** | Begins, ends, or cancels an operation in which the control is dragged. |
| **ShowWhatsThis** | Displays a selected topic in a Help file using the "What's This" pop-up provided by Help. |
| **Move** | Moves the position of the control. |
| **ZOrder** | Places the control at the front or back of the z-order within its graphical level. |
| **SetFocus** | Sets the focus to the control. |

The **Extender** object events and Visual Basic–specific extender events are listed following.

| **Event** | **Description** |
|---|---|
| DragDrop | Occurs when another control on the form is dropped on this control. |
| LostFocus | Occurs when the control loses focus. |
| DragOver | Occurs when another control on the form is dragged over this control. |
| GotFocus | Occurs when the control gets focus |

To locate more information about these methods, properties, and events, see Further Information on Streaming Video Services for the Client.

## BPCVid

[This is preliminary documentation and subject to change.]

The **BPCVid** class defines the Microsoft® ActiveX™ object which provides streaming video functionality. It is designed to be forward-compatible with the Microsoft® DirectShow™ version 1.0 ActiveX control.

**Note** Version 1.0 of Broadcast Architecture is not compatible with the DirectShow 1.0 ActiveX control.

For more information, see the following topics:

- **BPCVid** Properties
- **BPCVid** Methods
- **BPCVid** Events

## BPCVid Properties

[This is preliminary documentation and subject to change.]

The **BPCVid** object stores the following properties:

| **Property** | **Description** |
|---|---|
| **ClosedCaption** | A value that indicates whether closed captioning is turned on. |
| **Debug** | A value that indicates whether debugging information is displayed. |
| **DeviceCount** | The number of devices currently available. |
| **Devices** | A collection of the devices currently available on the user's computer. |
| **DisplayMode** | A value that indicates whether the current position in the video is displayed in number of frames displayed or time elapsed. |
| **Font** | The font used in the title window. |
| **LocaleID** | The locale identifier. This property supports the country codes of Microsoft® Windows® operating systems and determines which control character set to display, which date and currency formats to use, and so on. |
| **MovieWindowSetting** | A value that sizes the window displaying the images associated with a multimedia stream. |
| **StartTime** | The starting position in the multimedia stream. |
| **StopTime** | The ending position in the multimedia stream. |

The following properties are not stored locally in the **BPCVid** object; instead, they are passed to **BPCVid** object's corresponding **BPCDevices** object. However, you must set them using **BPCVid**. If you attempt to set them using a reference to **BPCDevices**, **BPCVid** is not updated, and future behavior is undefined.

| **Property** | **Description** |
|---|---|
| **ColorKey** | The color to use for color keying. |
| **HWnd** | A handle to the window hosting the Video control. |
| **Input** | The input device. |
| **Output** | The output device. |

| | |
|---|---|
| Priority | The priority of this application. This value is used to resolve device conflicts. |
| VideoOn | A value that indicates whether video is displayed on the control. |

The following properties of **BPCVid** are passed to current input device. However, you should set them in the **BPCVid** object instead of accessing the **BPCDevice** base object directly. Otherwise, the Video control is not properly updated, and future behavior is undefined.

| **Property** | **Description** |
|---|---|
| Balance | The current audio balance between left and right speakers. |
| CurrentPosition | The current position in the multimedia stream. |
| Duration | The duration of the multimedia stream. |
| ImageSourceHeight | The authored height of the source image.<br><br>This value does not change if the user resizes the control. |
| ImageSourceWidth | The authored width of the source image.<br><br>This value does not change if the user resizes the control. |
| Power | A value that indicates whether the device is currently turned on. |
| PrerollTime | A value that indicates the amount of time to allow a tape to roll before starting to record. |
| Rate | The rate of the multimedia stream. |
| Volume | The current audio volume level. |

The following properties of **BPCVid** are passed to the Visual Basic **Extender** object. To locate more information about **Extender** properties, see Further Information on Streaming Video Services for the Client.

| **Property** | **Description** |
|---|---|
| **Container** | Object that represents the visual container of the control. |
| **DragIcon** | Picture that specifies the icon to use when the control is dragged. |
| **DragMode** | **Integer** that specifies if the control automatically drags, or if the application using the control must call the **Drag** method. |
| **Height** | **Integer** that specifies the height of the control in the container's scale units. |

| | |
|---|---|
| **HelpContextID** | **Integer** that specifies the context identifier to use when the F1 key is pressed when the control has the focus. |
| **Index** | **Integer** that specifies the position in a control array this instance of the control occupies. |
| **Left** | **Integer** that specifies the position of the left edge of the control relative to the left edge of the container, specified in the container's scale units. |
| **Name** | **String** that contains the user-defined name of the control. |
| **Parent** | Object that represents the container of the control, such as a form in Visual Basic. |
| **TabIndex** | **Integer** that specifies the position of the control in the tab order of the controls in the container. |
| **TabStop** | Boolean value that specifies if the tab stops on the control. |
| **Tag** | **String** that contains a user-defined value. |
| **ToolTipText** | **String** that contains the text to be displayed when the cursor hovers above the control from more than a second. |
| **Top** | **Integer** that specifies the position of the upper edge of the control relative to the upper edge of the container, specified in the container's scale units. |
| **Visible** | Boolean value that specifies whether the control is visible. |
| **WhatThisHelpID** | Context identifier to use when the "What's This" pop-up is used to provide information on the control. |
| **Width** | Width of the control in the container's scale units. |

## BPCVid.ClosedCaption

[This is preliminary documentation and subject to change.]

The **ClosedCaption** property indicates whether closed captioning is turned on.

**Syntax**

```
object.ClosedCaption [ = boolean ]
```

### Parts

*object*
: Object expression that resolves to a **BPCVid** object.

*boolean*
: Boolean expression that is **True** if closed captioning is enabled and **False** if it is disabled.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCVid.Debug

[This is preliminary documentation and subject to change.]

The **Debug** property specifies whether debugging information is displayed on-screen.

### Syntax

```
object.Debug [ = boolean ]
```

### Parts

*object*
: Object expression that resolves to a **BPCVid** object.

*boolean*
: Boolean value that is **True** if debugging information is shown on-screen in the Video control and **False** if it is not.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCVid.DeviceCount

[This is preliminary documentation and subject to change.]

The **DeviceCount** property is a read-only property that returns the number of objects in the **BPCDevices** collection.

### Syntax

```
object.DeviceCount
```

### Parts

*object*
: Object expression that resolves to a **BPCVid** object.

### Settings

A **Long** that specifies the number of objects in the **Devices** collection.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

### See Also

**Devices**

# BPCVid.Devices

[This is preliminary documentation and subject to change.]

The **Devices** property is a read-only property that returns a **BPCDevices** collection of the available devices.

### Syntax

```
object.Devices
```

### Parts

*object*
  Object expression that resolves to a **BPCVid** object.

### Settings

A **BPCDevices** object.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCVid.DisplayMode

[This is preliminary documentation and subject to change.]

The **DisplayMode** property specifies the current position in the video is displayed in number of frames displayed or time elapsed.

### Syntax

```
object.DisplayMode [ = DisplayMode ]
```

### Parts

*object*
  Object expression that resolves to a **BPCVid** object.
*DisplayMode*
  Specifies the mode. The mode can be one of the following values:

| **Value** | **Meaning** |
|---|---|
| modeFrames | The current position is measured in frames. |
| modeTime | The current position is measured in time. |

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCVid.Font

[This is preliminary documentation and subject to change.]

The **Font** property specifies the font used in the title bar of the Video control.

### Syntax

```
object.Font [ = Font ]
```

### Parts

*object*
Object expression that resolves to a **BPCVid** object.
*Font*
An **IFontDisp** interface that specifies the font. To locate more information about **IFontDisp**, see Further Information on Streaming Video Services for the Client.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCVid.LocaleID

[This is preliminary documentation and subject to change.]

The **LocaleID** property sets or returns the identifier for the current locale.

### Syntax

```
object.LocaleID [ = long]
```

### Parts

*object*
    Object expression that resolves to a **BPCVid** object.
*long*
    **Long** that specifies the locale identifier.

### Remarks

This property supports the country codes of the Windows operating systems. The locale determines which control character set to display, which formats to use for dates and currency, and so on. To locate more information on locale identifiers, see Further Information on Streaming Video Services for the Client.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCVid.MovieWindowSetting

[This is preliminary documentation and subject to change.]

The **MovieWindowSetting** property sets or returns a value that sizes the window displaying the images associated with a multimedia stream. This property is not fully implemented. The size is always the specified window size adjusted down to the nearest 4/3 aspect ratio.

### Syntax

```
object.MovieWindowSetting [ = setting ]
```

### Parts

*object*
    Object expression that resolves to a **BPCVid** object.

*setting*

**Integer** or constant to specify the window size, as described in the following table.

| Value | Meaning |
|---|---|
| **movieDefaultSize** | Uses the default authored size. |
| **movieDoubleSize** | Increases the image to twice the authored size. |
| **movieFullScreen** | Projects the image onto a full-screen control window. |
| **movieHalfSize** | Sizes the image to fit half the screen. |
| **movieMaximizeSize** | Projects the image onto a maximized control window. |
| **moviePermitResizeNoRestrict** | Enables the user to resize the image without restriction. |
| **moviePermitResizeWithAspect** | Enables the user to resize the image, while maintaining a 4/3 aspect ratio. |

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

### See Also

**ImageSourceHeight**, **ImageSourceWidth**

## BPCVid.StartTime

[This is preliminary documentation and subject to change.]

The **StartTime** property sets or returns the starting position in the multimedia stream.

### Syntax

```
object.StartTime [ = double]
```

### Parts

*object*
: Object expression that resolves to a **BPCVid** object.

*double*
: Double that specifies the starting position of the multimedia stream. The default value is 0 (zero).

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

### See Also

**BPCDeviceBase.Duration**, **BPCVid.StopTime**

## BPCVid.StopTime

[This is preliminary documentation and subject to change.]

The **StopTime** property sets or returns the ending position in the multimedia stream.

### Syntax

```
object.StopTime [ = double]
```

### Parts

*object*
: Object expression that resolves to a **BPCVid** object.

*double*
: Double that specifies the ending position of the multimedia stream. The default value is the value of the **Duration** property.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

### See Also

**BPCDeviceBase.Duration**, **BPCVid.StartTime**

## BPCVid Methods

[This is preliminary documentation and subject to change.]

The following methods are implemented in the **BPCVid** object.

| **Method** | **Description** |
|---|---|
| **AboutBox** | Displays version and copyright information about the Video control. |
| **Close** | Sets all inputs and outputs to NULL. |

The following methods are implemented in the **BPCVid** object as wrappers for the equivalent methods in the **BPCDevices** object.

| **Method** | **Description** |
|---|---|
| **AutoScan** | Returns the signal strength of the channel. |
| **Tune** | Selects an input device based on the *tuning space* parameter, sets the current input device to the selected device, and tunes the current input device to the specified channel. |
| **TSDeviceCount** | Returns the number of devices available for a given tuning space. |

The following methods are implemented in the **BPCVid** object as wrappers for the equivalent methods in the **BPCDeviceBase** object.

| **Method** | **Description** |
|---|---|
| **Refresh** | Forces an update to the current window size, position, and visibility. |

The following methods are implemented in the **BPCVid** object as wrappers for the equivalent methods in the Visual Basic **Extender** object.

| Method | Description |
|---|---|
| **Drag** | Begins, ends, or cancels an operation in which the control is dragged. |
| **ShowWhatsThis** | Displays a selected topic in a Help file using the "What's This" pop-up provided by Help. |
| **Move** | Moves the position of the control. |
| **ZOrder** | Places the control at the front or back of the z-order within its graphic level. |
| **SetFocus** | Set the focus to the control. |

## BPCVid.AboutBox

[This is preliminary documentation and subject to change.]

The **AboutBox** method displays version and copyright information about the control.

### Syntax

```
object.AboutBox
```

### Parameters

*object*
Object expression that resolves to a **BPCVid** object.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCVid.Close

[This is preliminary documentation and subject to change.]

The **Close** method sets all inputs and outputs to NULL.

### Syntax

```
object.Close
```

### Parameters

*object*

Object expression that resolves to a **BPCVid** object.

### Remarks

This method does not affect device tuning.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCVid Events

[This is preliminary documentation and subject to change.]

The **DBPCVidEvents** interface is implemented on the **BPCVid** object and is used to send event notifications to the application or container hosting the Video control.

The following events are sent by the Video control.

| Event | Description |
|---|---|
| Click | A user clicked in the control. |
| DblClick | A user double-clicked in the control. |
| Error | An error has occurred in the control or in the control's input or output device. |
| ErrorMessage | An error has occurred. This event is not implemented. |
| GotControl | The control receiving this notification has requested a device and now has the highest priority for that device. |
| KeyDown | The user pushed a key down. |

| | |
|---|---|
| KeyPress | The user pressed a key. |
| KeyUp | The user released a key. |
| LostControl | An application with higher priority than the current application requested the control's input or output device. The instance of the Video control receiving this event must release the device. |
| MouseDown | The user clicked a mouse button over the Video control. |
| MouseMove | The user moved the mouse over the Video control. |
| MouseUp | The user has released a mouse button over the Video control. |

## BPCVid.Click

[This is preliminary documentation and subject to change.]

The Click event occurs when the user clicks the Video control with the left or right mouse button.

### Syntax

```
Private Sub object_Click
```

### Parameters

*object*
Object expression that resolves to a **BPCVid** object.

### Remarks

To distinguish between the left, right, and middle mouse buttons, use the **BPCVid.**MouseDown and **BPCVid.**MouseUp events.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCVid.DblClick

[This is preliminary documentation and subject to change.]

The DblClick event occurs when the user double-clicks the Video control.

### Syntax

```
Private Sub object_DblClick
```

### Parameters

*object*
    Object expression that resolves to a **BPCVid** object.

### Remarks

To distinguish between the left, right, and middle mouse buttons, use the **BPCVid.MouseDown** and **BPCVid.MouseUp** events.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCVid.Error

[This is preliminary documentation and subject to change.]

The Error event occurs when an asynchronous error occurs in the Video control or the control's input or output device.

### Syntax

```
Private Sub object_Error(Number As Integer, _
 ByVal Description As String, ByVal SCode As Integer, _
 ByVal Source As String, HelpFile As String, _
 HelpContext As Long, CancelDisplay As Boolean)
```

**Parameters**

*object*
Object expression that resolves to a **BPCVid** object.
*Number*
**Integer** that contains the low WORD of the SCode parameter.
*Description*
**String** describing the error that occurred.
*SCode*
Error code.
*Source*
**String** containing the control's name.
*HelpFile*
**String** containing the Help file name.
*HelpContext*
A **Long** that indicates the Help context.
*CancelDisplay*
Value that may be set by the client to cancel the default error messages.

**Remarks**

Broadcast Architecture does not currently generate asynchronous errors.

The Error event occurs when the Video control reports an error during playback. By default, the control displays a message box containing the description string. To avoid displaying this box, set the *CancelDisplay* parameter of the Error event to **False**.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCVid.ErrorMessage

[This is preliminary documentation and subject to change.]

The **ErrorMessage** event passes an error message to the container application hosting the Video control. This event is not implemented.

**Syntax**

```
Private Sub object_ErrorMessage(iMessage As Long, Text As String)
```

### Parameters

*object*
: Object expression that resolves to a **BPCVid** object.

*iMessage*
: **Long** that indicates the error number.

*Text*
: **String** that describes the error.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCVid.GotControl

[This is preliminary documentation and subject to change.]

The GotControl event notifies an instance of the Video control that the instance has control of a device.

### Syntax

```
Private Sub object_GotControl
```

### Parameters

*object*
: Object expression that resolves to a **BPCVid** object.

### Remarks

This event occurs if the control requests a device that is not currently being used by an application with higher priority. If two devices have the same priority, the current control retains the device.

Your application must wait until it receives a GotControl event before it uses a device. Otherwise, applications currently using the device do not release the device, and unexpected behavior can occur.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

### See Also

**LostControl**

## BPCVid.KeyDown

[This is preliminary documentation and subject to change.]

The KeyDown event occurs when the user presses a key while the Video control has the focus.

### Syntax

```
Private Sub object_KeyDown(keycode As Integer, shift As Integer)
```

### Parameters

*object*
Object expression that resolves to a **BPCVid** object.

*keycode*
Key code, such as vbKeyF1 (the F1 key) or vbKeyHome (the HOME key). To specify key codes, use the constants in the Visual Basic object library in the Visual Basic Object Browser.

*shift*
**Integer** that corresponds to the state of the shift, ctrl, and alt keys at the time of the event. The *shift* parameter is a bit field with the least-significant bits corresponding to the shift key (bit 0), the ctrl key (bit 1), and the alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both ctrl and alt are pressed, the value of *shift* is 6.

### Remarks

To locate more information on the Visual Basic Object Browser, see Further Information on Streaming Video Services for the Client.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

### See Also

**BPCVid.KeyPress**, **BPCVid.KeyUp**

## BPCVid.KeyPress

[This is preliminary documentation and subject to change.]

The KeyPress event occurs when the user presses and releases an ANSI key while the Video control has focus.

### Syntax

```
Private Sub object_KeyPress(keyascii As Integer)
```

### Parameters

*object*
Object expression that resolves to a **BPCVid** object.
*keyascii*
**Integer** that returns a standard numeric ANSI keycode. Changing *keyascii* to 0 cancels the key stroke so the object is not assigned a character.

### Remarks

A KeyPress event can involve any printable keyboard character, the ctrl key combined with a character from the standard alphabet or one of a few special characters, and the enter or backspace key.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

### See Also

**BPCVid.KeyDown**, **BPCVid.KeyUp**

# BPCVid.KeyUp

[This is preliminary documentation and subject to change.]

The KeyUp event occurs when the user releases a pressed key while the Video control has the focus.

### Syntax

```
Private Sub object_KeyUp(keycode As Integer, shift As Integer)
```

### Parameters

*object*
: Object expression that resolves to a **BPCVid** object.

*keycode*
: Key code, such as vbKeyF1 (the F1 key) or vbKeyHome (the HOME key). To specify key codes, use the constants in the Visual Basic object library in the Visual Basic Object Browser.

*shift*
: **Integer** that corresponds to the state of the shift, ctrl, and alt keys at the time of the event. The *shift* parameter is a bit field with the least-significant bits corresponding to the shift key (bit 0), the ctrl key (bit 1), and the alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both ctrl and alt are pressed, the value of *shift* is 6.

### Remarks

To locate more information on the Visual Basic Object Browser, see Further Information on Streaming Video Services for the Client.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

### See Also

**BPCVid.KeyDown**, **BPCVid.KeyPress**

# BPCVid.LostControl

[This is preliminary documentation and subject to change.]

The LostControl event notifies an instance of the Video control that it has lost control of the input device.

**Syntax**

```
Private Sub object_LostControl
```

**Parameters**

*object*
Object expression that resolves to a **BPCVid** object.

**Remarks**

This event occurs when an application with a higher priority requests the device. Your application must honor this request and immediately release the device. If an application continues to manipulate a device after it has received a LostControl notification, unexpected behavior can occur.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

**See Also**

**BPCVid.**GotControl

# BPCVid.MouseDown

[This is preliminary documentation and subject to change.]

The MouseDown event occurs when the user presses a mouse button while the Video control has focus.

## Syntax

```
Private Sub object_MouseDown(button As Integer, shift As Integer, _
 x As OLE_XPOS_PIXELS, y As OLE_YPOS_PIXELS)
```

## Parameters

*object*
Object expression that resolves to a **BPCVid** object.
*button*
**Integer** that on return identifies the button pressed to cause the event. The *button* parameter is a bit field with bits corresponding to the left button (bit 0), right button (bit 1), and middle button (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Only one of the bits is set, indicating the button that caused the event.
*shift*
**Integer** that corresponds to the state of the shift, ctrl, and alt keys at the time of the event. The *shift* parameter is a bit field with the least-significant bits corresponding to the shift key (bit 0), the ctrl key (bit 1), and the alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both ctrl and alt are pressed, the value of *shift* is 6.
*x*
Number that on return specifies the x-coordinate of the current mouse pointer location.
*y*
Number that on return specifies the y-coordinate of the current mouse pointer location.

## QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## See Also

**BPCVid.MouseMove**, **BPCVid.MouseUp**

# BPCVid.MouseMove

[This is preliminary documentation and subject to change.]

The MouseMove event occurs when the user moves the mouse.

### Syntax

```
Private Sub object_MouseMove(button As Integer, shift As Integer, _
 x As OLE_XPOS_PIXELS, y As OLE_YPOS_PIXELS)
```

### Parameters

*object*
    Object expression that resolves to a **BPCVid** object.
*button*
    **Integer** that on return identifies the button pressed to cause the event. The *button* parameter is a bit field with bits corresponding to the left button (bit 0), right button (bit 1), and middle button (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Only one of the bits is set, indicating the button that caused the event.
*shift*
    **Integer** that corresponds to the state of the shift, ctrl, and alt keys at the time of the event. The *shift* parameter is a bit field with the least-significant bits corresponding to the shift key (bit 0), the ctrl key (bit 1), and the alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both ctrl and alt are pressed, the value of *shift* is 6.
*x*
    Number that on return specifies the x-coordinate of the current mouse pointer location.
*y*
    Number that on return specifies the y-coordinate of the current mouse pointer location.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

### See Also

**BPCVid.MouseDown**, **BPCVid.MouseUp**

# BPCVid.MouseUp

[This is preliminary documentation and subject to change.]

The MouseUp event occurs when the user releases a pressed mouse button.

### Syntax

```
Private Sub object_MouseUp(button As Integer, shift As Integer, _
 x As OLE_XPOS_PIXELS, y As OLE_YPOS_PIXELS)
```

### Parameters

*object*
: Object expression that resolves to a **BPCVid** object.

*button*
: **Integer** that on return identifies the button pressed to cause the event. The *button* parameter is a bit field with bits corresponding to the left button (bit 0), right button (bit 1), and middle button (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Only one of the bits is set, indicating the button that caused the event.

*shift*
: **Integer** that corresponds to the state of the shift, ctrl, and alt keys at the time of the event. The *shift* parameter is a bit field with the least-significant bits corresponding to the shift key (bit 0), the ctrl key (bit 1), and the alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both ctrl and alt are pressed, the value of *shift* is 6.

*x*
: Number that on return specifies the x-coordinate of the current mouse pointer location.

*y*
: Number that on return specifies the y-coordinate of the current mouse pointer location.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

### See Also

**BPCVid.MouseDown**, **BPCVid.MouseMove**

## BPCDevices

[This is preliminary documentation and subject to change.]

The **BPCDevices** object is a collection of all the devices currently available on the computer. In other words, **BPCDevices** is a collection of **BPCDeviceBase** objects.

The Video Access server creates an instance of one of these objects for each **BPCVid** object.

For more information, see the following topics:

- **BPCDevices** Properties
- **BPCDevices** Methods

## BPCDevices Properties

[This is preliminary documentation and subject to change.]

The **BPCDevices** object has the following properties.

| **Property** | **Description** |
|---|---|
| **ColorKey** | The color to use for color keying. |
| **Count** | The number of devices currently available. |
| **HWnd** | A reference to the handle of the window hosting the Video control. |
| **Input** | The input device. |
| **LCID** | The locale identifier. This property supports the country codes of Windows operating systems and determines which control character set to display, which date and currency formats to use, and so on. |
| **Notify** | A reference to the interface to send internal events to. |
| **Output** | The output device. |
| **Priority** | The application priority. This value is used to resolve device contention. |
| **VideoOn** | A value that indicates whether video is displayed through the control. |

## BPCDevices.ColorKey

[This is preliminary documentation and subject to change.]

The **ColorKey** property specifies the color to use for color keying. This property is not fully implemented. Currently, the **ColorKey** value is always set to magenta (0x00FF00FF).

### Syntax

```
object.ColorKey [ = colorval ]
```

### Parts

*object*
Object expression that resolves to a **BPCVid** or **BPCDevices** object.
*colorval*
A **Long** or constant that determines the color to use for color keying. The following table lists and describes the possible values for the *colorval* parameter.

| Value | Meaning |
|---|---|
| Normal RGB colors | Colors specified by using the **Color** palette or programmatically by using the **RGB** or **QBColor** functions. For more information, see Further Information on Streaming Video Services for the Client. |
| System default colors | Colors specified by system color constants listed in the object library in the Visual Basic Object Browser. As needed, Windows substitutes the user's color choices for default colors as specified in the Control Panel settings. |

### Remarks

The valid range for a normal RGB color is 0 to 16,777,215 (&HFFFFFF). The high byte of a number in this range equals zero; the lower three bytes, from least to most significant byte, determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number between 0 and 255 (&HFF). If the high byte isn't zero, Visual Basic uses the system colors, as defined in the user's Control Panel settings and by constants listed in the object library in the Object Browser.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCDevices.Count

[This is preliminary documentation and subject to change.]

The **Count** property returns the number of devices currently available on the user's computer. This property is read-only.

### Syntax

```
object.Count
```

### Parts

*object*
Object expression that resolves to a **BPCDevices** object.

### Settings

A **Long** that indicates the number of **BPCDeviceBase** objects in the **BPCDevices** collection.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDevices.HWnd

[This is preliminary documentation and subject to change.]

The **HWnd** property contains a reference to the window hosting the Video control.

### Syntax

```
object.HWnd [ = lHwnd]
```

### Parts

*object*
Object expression that resolves to a **BPCVid** or **BPCDevices** object.
*lHwnd*
A **Long** that specifies the handle of the host window.

### Remarks

Windows identifies each form and control in an application by assigning it a handle, or HWnd. The **HWnd** property is used with Microsoft® Win32® API calls, most often as a parameter.

Because Windows may change the value of **HWnd**, your application must not change the **HWnd** value or store it in a variable. To locate more information on using **HWnd**, see Further Information on Streaming Video Services for the Client.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDevices.Input

[This is preliminary documentation and subject to change.]

The **Input** property sets or returns information on the device supplying input to the Video control.

### Syntax

```
object.Input [ = deviceobject]
```

### Parts

*object*
Object expression that resolves to a **BPCVid** or **BPCDevices** object.
*deviceobject*
Object expression that resolves to a **BPCDeviceBase** object.

### Remarks

Once you assign a **BPCDeviceBase** object to the **Input** property, the media stream from the input device is displayed by the control. To suppress the display, set the **BPCDevices.VideoOn** property to **False**. Alternatively, you can set the **Visible** property to **False** to hide the entire window. The **Visible** property is implemented by the **Extender** object; for more information, see Visual Basic Extender Object.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

### See Also

**BPCDeviceBase**, **BPCDevices.Output**, **BPCDevices.VideoOn**

## BPCDevices.LCID

[This is preliminary documentation and subject to change.]

The **LCID** property returns the locale identifier.

### Syntax

```
object.LCID
```

### Parts

*object*
Object expression that resolves to a **BPCDevices** object.

### Settings

A **Long** that indicates the locale identifier. Your application must not change this value.

### Remarks

This property supports the country codes of Windows operating systems and determines which control character set to display, which date and currency formats to use, and so on. To locate more information on locale identifiers, see Further Information on Streaming Video Services for the Client.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDevices.Notify

[This is preliminary documentation and subject to change.]

The **Notify** property sets the instance of the Video control to receive event notifications. Your application must not change this value.

### Syntax

```
object.Notify [ = oDevice ]
```

### Parts

*object*
Object expression that resolves to a **BPCDevices** object.

*oDevice*
**BPCDevices** object.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDevices.Output

[This is preliminary documentation and subject to change.]

The **Output** property sets or returns the device to which the Video control sends output.

### Syntax

```
object.Output[ = deviceobject]
```

### Parts

*object*
    Object expression that resolves to a **BPCVid** or **BPCDevices** object.
*deviceobject*
    Object expression that resolves to a **BPCDeviceBase** object.

**Remarks**

When you set this property to a valid output device, the media stream coming from the **Input** property is:

- Sent to this device.
- Displayed in the control window.

To suppress this output, set this property to the value Nothing or to the **BPCDeviceBase** object named **Null** in the **BPCDevices** collection.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

**See Also**

**BPCDevices.Input**

# BPCDevices.Priority

[This is preliminary documentation and subject to change.]

The **Priority** property sets or returns a value that defines the priority of an instance of the Video control in the system.

**Syntax**

```
object.Priority [ = long]
```

**Parts**

*object*
    Object expression that resolves to a **BPCVid** or **BPCDevices** object.

*long*

A **Long** that determines priority for this control instance.

**Remarks**

Use this property to surrender or require control of a resource. To relinquish any claim to a resource, set the **BPCDevices.Input** or **BPCDevices.Output** property to Nothing.

This property is used to resolve device conflicts. When two control instances are in contention for a device, control is given to the application with higher priority. If both instances have equal priority, the instance that currently has control of the device retains control.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCDevices.VideoOn

[This is preliminary documentation and subject to change.]

The **VideoOn** property sets or returns a value indicating whether video is displayed on the control.

**Syntax**

```
object.VideoOn [ = state ]
```

**Parts**

*object*

Object expression that resolves to a **BPCVid** or **BPCDevices** object.

*state*

Boolean expression that is **True** if multimedia data from the device indicated by the **BPCDevices.Input** property is displayed by the Video control, and **False** if it is not displayed. **True** is the default.

**Remarks**

You can use this property to turn off video when your application needs to pass the contents of the current input device to the output device without displaying anything on-screen. For example, you can use **VideoOn** if your application is scheduling a recording in the background.

To route media data from the device indicated by **Input** to the device indicated by the **BPCDevices.Output** property without displaying the media data, set **VideoOn** to **False**.

To play audio-only content, set **VideoOn** to **True** and set the **BPCDevices.Height** and **BPCDevices.Width** properties to zero.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

**See Also**

**BPCDevices.Input**

## BPCDevices Methods

[This is preliminary documentation and subject to change.]

| Method | Description |
|---|---|
| **AutoScan** | Returns the signal strength of the current channel. |
| **Item** | Returns an item from the **BPCDevices** collection. |
| **TSDeviceCount** | Returns the number of devices associated with a tuning space. |
| **Tune** | Tunes the current input device. |

# BPCDevices.AutoScan

[This is preliminary documentation and subject to change.]

The **AutoScan** method returns the signal strength of the current channel.

**Syntax**

```
object.AutoScan
```

**Parameters**

*object*
: Object expression that resolves to a **BPCDevices** object.

**Return Values**

A **Long** indicating the signal strength. The value returned depends on the device. If a channel is present but the device does not support signal strength measurements, this method returns 1.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDevices.Item

[This is preliminary documentation and subject to change.]

The **Item** method returns a reference to the specified **BPCDeviceBase** object in the **BPCDevices** collection. This property is read-only.

**Syntax**

```
object.Item(v)
```

**Parameters**

*object*
: Object expression that resolves to a **BPCDevices** object.

*v*
: VT_BSTR value that contains the device name of the **BPCDeviceBase** object.

**Return Values**

The specified **BPCDeviceBase** object.

**QuickInfo**

**Windows NT:** Unsupported.

**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDevices.TSDeviceCount

[This is preliminary documentation and subject to change.]

The **TSDeviceCount** method returns the number of devices for a specified tuning space.

### Syntax

```
object.TSDeviceCount(lTuningSpace)
```

### Parameters

*object*
Object expression that resolves to a **BPCVid** or **BPCDevices** object.
*lTuningSpace*
A **Long** that specifies the tuning space identifier. This identifier is the same value as that specified in the Guide database.

### Return Values

A **Long** that indicates the number of devices.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDevices.Tune

[This is preliminary documentation and subject to change.]

The **Tune** method tunes the current input device to the specified channel.

### Syntax

```
object.Tune(lTuningSpace, Channel, VideoSubchannel, AudioSubchannel)
```

### Parameters

*object*
    Object expression that resolves to a **BPCVid** or **BPCDevices** object.
*lTuningSpace*
    A **Long** that specifies the tuning space identifier. This identifier is the same value as that specified in the Guide database.
*Channel*
    A **Long** that specifies the channel number.
*VideoSubchannel*
    A **Long** that specifies the video subchannel.
*AudioSubchannel*
    A **Long** that specifies the audio subchannel.

### Remarks

You can use either **BPCVid.Tune** method or **BPCVid.Input.Channel** property to set the channel displayed by the Video control. If you are simply trying to tune the Video control to a particular channel and do not care which device provides the channel you can use the **Tune** method. This method automatically sets **BPCVid.Input** to the appropriate device, as identified by *ITuningSpace*. However, if it is important that your application set the **Input** property to a particular device, you can use the **BPCVid.Input.Channel** syntax.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDevices Events

[This is preliminary documentation and subject to change.]

The following events are sent by the Video Access server. The Video control passes these on to the application that is hosting it.

| **Event** | **Description** |
|---|---|
| GotControl | The control receiving this notification has requested a device, and now has the highest priority. |
| LostControl | An application with higher priority requested the control's input or output device. The control receiving this notification is losing control. |

## BPCDeviceBase

[This is preliminary documentation and subject to change.]

The **BPCDeviceBase** object contains information about an individual device on the user's computer. The **BPCDevices** collection enumerates these objects.

For more information, see the following topics:

- **BPCDeviceBase** Properties
- **BPCDeviceBase** Methods

**Remarks**

There is no tuning space property for **BPCDeviceBase**. Instead the tuning space value associated with a particular device is stored as a registry key of the form, \\HKLM\Software\Microsoft\Tv Services\Tuning Spaces\<*tuning_space*>, where <*tuning_space*> is a number that is the identifier of the tuning space.

A single device can provide multiple tuning spaces. For example an analog tuner might provide both the cable and antenna broadcast tuning spaces.

### BPCDeviceBase Properties

[This is preliminary documentation and subject to change.]

The **BPCDeviceBase** class defines the following properties:

| **Property** | **Description** |
|---|---|
| **AudioFrequency** | The frequency of the audio stream. |
| **AudioSubchannel** | The audio subchannel. |
| **Balance** | The current audio balance between left and right speakers. |
| **Channel** | The channel that the device is currently tuned to. |
| **CountryCode** | The international dialing code that specifies the geographical location of the computer. |
| **CurrentPosition** | The current position in the multimedia stream. |
| **DefaultAudioType** | The default type for the audio stream. |
| **DefaultVideoType** | The default type for the video stream. |
| **Duration** | The duration of the multimedia stream |
| **HasChannel** | A value that indicates whether the device supports channels. |
| **ImageSourceHeight** | The height of the source image. |
| **ImageSourceWidth** | The width of the source image. |
| **IsInput** | A value that indicates whether the device is set as the input device. |
| **IsOutput** | A value that indicates whether the device is set as the output device. |
| **Name** | A unique name for the device. |
| **OverScan** | The percent of pixels to trim from the edge of the video screen. |
| **Power** | The power state of the device, either on or off. |
| **PrerollTime** | The lead time in seconds for a tape or other recording device to start before it begins recording. |
| **ProdName** | The product name of the device. This name does not have to be unique. |
| **Rate** | The playback speed multiplier relative to the normal playback rate. |
| **UserName** | The user-specified name for the device. This property is not currently implemented. |
| **VideoFrequency** | The frequency of the video stream. |
| **VideoSubchannel** | The video subchannel. |
| **Volume** | The volume setting for the device. |

# BPCDeviceBase.AudioFrequency

[This is preliminary documentation and subject to change.]

The **AudioFrequency** property specifies the frequency of the audio signal. This property is read-only.

### Syntax

```
object.AudioFrequency
```

### Parts

*object*
Object expression that resolves to a **BPCDeviceBase** object.

### Settings

A **Long** that specifies the frequency, in hertz.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCDeviceBase.AudioSubchannel

[This is preliminary documentation and subject to change.]

The **AudioSubchannel** property sets or retrieves the audio stream subchannel of a device.

### Syntax

```
object.AudioSubchannel [ = lSubChannel ]
```

### Parts

*object*
Object expression that resolves to a **BPCDeviceBase** object.

*lSubChannel*
A **Long** that specifies the subchannel.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCDeviceBase.Balance

[This is preliminary documentation and subject to change.]

The **Balance** property sets or retrieves a value that controls the multimedia stream's audio balance between left and right speakers.

**Syntax**

```
object.Balance [ = long]
```

**Parts**

*object*
Object expression that resolves to a **BPCVid** or **BPCDeviceBase** object.
*long*
A **Long** that specifies the balance value. The number must be from –10,000 through +10,000. The default value is zero.

**Remarks**

The default value of zero indicates equal balance between left and right.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

**See Also**

**BPCDeviceBase.Volume**

# BPCDeviceBase.Channel

[This is preliminary documentation and subject to change.]

The **Channel** property sets or returns the channel that the device is currently tuned to.

### Syntax

```
object.Channel [ = lChannel ]
```

### Parts

*object*
Object expression that resolves to a **BPCDeviceBase** object.
*lChannel*
Channel number tuned to.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCDeviceBase.CountryCode

[This is preliminary documentation and subject to change.]

The **CountryCode** property enables you to set or retrieve the country identifier for the device.

### Syntax

```
object.CountryCode [ = lCode ]
```

### Parts

*object*
: Object expression that resolves to a **BPCDeviceBase** object.

*lCode*
: A **Long** that specifies the country code.

### Remarks

This property is an international dialing code that specifies the geographical location of the computer. This location information is used to set the frequency and format of analog video — for example, to choose between *NTSC*, *PAL*, and *SECAM*. This property is only available for analog tuners.

To locate more information on this property, see Further Information on Streaming Video Services for the Client.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDeviceBase.CurrentPosition

[This is preliminary documentation and subject to change.]

The **CurrentPosition** property sets or retrieves the current position within the multimedia stream.

### Syntax

```
object.CurrentPosition [ = double]
```

### Parts

*object*
: Object expression that resolves to a **BPCVid** or **BPCDeviceBase** object.

*double*
: **Double** variable that designates the new position within the stream.

### Remarks

The new value must be within the range specified by the properties **BPCVid.StartTime** and **BPCVid.StopTime**.

Setting **CurrentPosition** at run time is similar to performing a seek operation. Setting this property changes the position to the specified point in the multimedia stream.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDeviceBase.DefaultAudioType

[This is preliminary documentation and subject to change.]

The **DefaultAudioType** property specifies the default audio type for the device.

### Syntax

```
object.DefaultAudioType [ = lDefAudioType ]
```

### Parts

*object*
Object expression that resolves to a **BPCDeviceBase** object.
*lDefAudioType*
A **Long** that indicates the default audio type. The audio types are device-specific types that allow the tuner to choose an appropriate subchannel if a default subchannel of –1 is specified. These types are defined in the Stream Type table of the Guide database.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDeviceBase.DefaultVideoType

[This is preliminary documentation and subject to change.]

The **DefaultVideoType** property specifies the default video type for this device.

### Syntax

```
object.DefaultVideoType [ = lDefaultVideoType ]
```

### Parts

*object*
Object expression that resolves to a **BPCDeviceBase** object.
*lDefaultVideoType*
A **Long** that indicates the default video type. The video types are device-specific types that allow the tuner to choose an appropriate subchannel if a default subchannel of –1 is specified. These types are defined in the Stream Type table of the Guide database.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCDeviceBase.Duration

[This is preliminary documentation and subject to change.]

The **Duration** property returns the duration of the multimedia stream in seconds. This is a read-only property.

### Syntax

```
object.Duration
```

### Parts

*object*
Object expression that resolves to a **BPCVid** or **BPCDeviceBase** object.

### Return Values

A **Double** that gives the duration in seconds.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDeviceBase.HasChannel

[This is preliminary documentation and subject to change.]

The **HasChannel** property indicates whether the device supports channels. This property is read-only.

### Syntax

```
object.HasChannel
```

### Parts

*object*
Object expression that resolves to a **BPCDeviceBase** object.

### Settings

A Boolean value that is **True** if the device supports channels and **False** if it does not.

### Remarks

A television tuner card, for example, has a **HasChannel** property of **True**.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDeviceBase.ImageSourceHeight

[This is preliminary documentation and subject to change.]

The **ImageSourceHeight** property contains the height of the source image. This property is read-only.

### Syntax

```
object.ImageSourceHeight
```

### Parts

*object*
Object expression that resolves to a **BPCVid** or **BPCDeviceBase** object.

### Settings

A **Long** that represents the height of the source image.

### Remarks

This is a read-only property. The value of this property is independent of the projected image size, which is determined by the **BPCVid.MovieWindowSetting** property.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

### See Also

**BPCDeviceBase.ImageSourceWidth**, **BPCVid.MovieWindowSetting**

## BPCDeviceBase.ImageSourceWidth

[This is preliminary documentation and subject to change.]

The **ImageSourceWidth** property contains the authored width of the source image. This property is read-only.

### Syntax

```
object.ImageSourceWidth
```

### Parts

*object*
Object expression that resolves to a **BPCVid** or **BPCDeviceBase** object.

### Settings

A **Long** that represents the width of the source image.

### Remarks

This is a read-only property. The value of this property is independent of the projected image size, which is determined by the **BPCVid.MovieWindowSetting** property.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

### See Also

**BPCDeviceBase.ImageSourceHeight**, **BPCVid.MovieWindowSetting**

## BPCDeviceBase.IsInput

[This is preliminary documentation and subject to change.]

The **IsInput** property indicates whether this device is the input device for the Video control. This property is read-only.

### Syntax

```
object.IsInput
```

### Parts

*object*
 Object expression that resolves to a **BPCDeviceBase** object.

**Settings**

A Boolean expression that is **True** if the device is the input device, and **False** if it is not.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCDeviceBase.IsOutput

[This is preliminary documentation and subject to change.]

The **IsOutput** property indicates whether this device is the output device for the Video control. This property is read-only.

**Syntax**

```
object.IsOutput
```

**Parts**

*object*
 Object expression that resolves to a **BPCDeviceBase** object.

**Settings**

A Boolean expression that is **True** if the device is the output device, and **False** if it is not.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDeviceBase.Name

[This is preliminary documentation and subject to change.]

The **Name** property returns the name of the current instance of the Video control. This property is read-only.

### Syntax

```
object.Name
```

### Parts

*object*
Object expression that resolves to a **BPCDeviceBase** object.

### Settings

**String** expression that identifies the control.

### Remarks

This is a read-only property at run time. This property is the name given to the instance when it is created. This name is extended to the control when used in Visual Basic, and it may or may not be supported in other ActiveX control containers.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDeviceBase.OverScan

[This is preliminary documentation and subject to change.]

The **OverScan** property sets the percentage of pixels to crop from the edges of the video picture.

### Syntax

```
object.OverScan [ = lCutPercentage ]
```

### Parts

*object*
Object expression that resolves to a **BPCDeviceBase** object.
*lCutPercentage*
A **Long** that contains the percent of pixels that should be trimmed from the edges of the video display. For example, if you set this value to 5 on a 100 x 100 pixel image, 5 percent of the pixels (in this case 5 pixels) is trimmed from each edge. The default value is 4.

### Remarks

Setting this property enables your application to display on a computer precisely the same image a viewer sees on a television set.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCDeviceBase.Power

[This is preliminary documentation and subject to change.]

The **Power** property sets or returns a value that controls whether an external device has power on or not.

### Syntax

```
object.Power [ = state ]
```

### Parts

*object*
Object expression that resolves to a **BPCVid** or **BPCDeviceBase** object.
*state*

Boolean expression that is **True** if the device is on and **False** if it is off.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDeviceBase.PrerollTime

[This is preliminary documentation and subject to change.]

The **PrerollTime** property specifies the time, in seconds, that a tape or other recording device should run before the multimedia stream starts.

**Syntax**

```
object.PrerollTime [ = dRollTime ]
```

**Parts**

*object*
Object expression that resolves to a **BPCVid** or **BPCDeviceBase** object.
*dRollTime*
**Double** that specifies the number of seconds the tape should run before the stream starts.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDeviceBase.ProdName

[This is preliminary documentation and subject to change.]

The **ProdName** property contains the product name of the device. This property is read-only.

### Syntax

```
object.ProdName
```

### Parts

*object*
: Object expression that resolves to a **BPCDeviceBase** object.

### Settings

**String** that contains the product name.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDeviceBase.Rate

[This is preliminary documentation and subject to change.]

The **Rate** property sets or returns the playback rate for the multimedia stream relative to the usual playback rate.

### Syntax

```
object.Rate [ = double ]
```

### Parts

*object*
: Object expression that resolves to a **BPCVid** or **BPCDeviceBase** object.

*double*
: **Double** that represents the playback rate. This value is a multiplier value that allows the stream to be played in slow motion or in fast motion. The default value of 1.0 indicates the usual speed (the authored speed). The audio track becomes difficult to understand at rates lower than 0.5

and higher than 1.5.

### Remarks

To locate more information on this property, see Further Information on Streaming Video Services for the Client.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDeviceBase.UserName

[This is preliminary documentation and subject to change.]

The **UserName** property enables you to set or retrieve the user name for this device.

### Syntax

```
object.UserName [ = String ]
```

### Parts

*object*
Object expression that resolves to a **BPCDeviceBase** object.
*String*
**String** that contains the user name.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCDeviceBase.VideoFrequency

[This is preliminary documentation and subject to change.]

The **VideoFrequency** property specifies the frequency of the video transmission. This property is read-only.

### Syntax

```
object.VideoFrequency
```

### Parts

*object*
Object expression that resolves to a **BPCDeviceBase** object.

### Settings

A **Long** that specifies the frequency, in hertz.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

# BPCDeviceBase.VideoSubchannel

[This is preliminary documentation and subject to change.]

The **VideoSubchannel** property enables you to set or retrieve the subchannel for the video stream of this device.

### Syntax

```
object.VideoSubchannel [ = lSubChannel ]
```

### Parts

*object*
Object expression that resolves to a **BPCDeviceBase** object.
*lSubChannel*
A **Long** that specifies the subchannel.

**Remarks**

The subchannel values are set in the Stream table of the Guide database.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDeviceBase.Volume

[This is preliminary documentation and subject to change.]

The **Volume** property sets or retrieves a value that controls the loudness of the multimedia stream.

**Syntax**

```
object.Volume [ = long ]
```

**Parts**

*object*
Object expression that resolves to a **BPCVid** or **BPCDeviceBase** object.
*long*
A **Long** that specifies the audio volume. The possible values range from –10,000 to 0. The default value, zero, represents full volume. The value –10,000 represents minimum volume.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

### See Also

**BPCDeviceBase.Balance**

## BPCDeviceBase Methods

[This is preliminary documentation and subject to change.]

The **BPCDeviceBase** object has the following methods.

| **Method** | **Description** |
|---|---|
| **ChannelAvailable** | Determines if the specified channel has a valid signal. |
| **DisplayConfigDialog** | Displays a dialog box for configuration of devices. |
| **Refresh** | Refreshes the device. |

# BPCDeviceBase.ChannelAvailable

[This is preliminary documentation and subject to change.]

The **ChannelAvailable** method returns a value indicating whether the specified channel has a valid signal.

### Syntax

```
object.ChannelAvailable(nChannel, SignalStrength)
```

### Parameters

*object*
Object expression that resolves to a **BPCDeviceBase** object.
*nChannel*
A **Long** that specifies the channel number.
*SignalStrength*
A **Long** that receives the signal strength.

### Return Values

A Boolean expression that is **True** if the specified channel has a valid signal and **False** if it does not.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDeviceBase.DisplayConfigDialog

[This is preliminary documentation and subject to change.]

The **DisplayConfigDialog** method displays a dialog box that enables the user to configure the device.

**Syntax**

```
object.DisplayConfigDialog
```

**Parameters**

*object*
Object expression that resolves to a **BPCDeviceBase** object.

**QuickInfo**

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCDeviceBase.Refresh

[This is preliminary documentation and subject to change.]

The **Refresh** method refreshes the device.

### Syntax

```
object.Refresh
```

### Parameters

*object*
Object expression that resolves to a **BPCVid** or **BPCDeviceBase** object.

### Remarks

This method forces an update to the current window size, position, and visibility.

### QuickInfo

**Windows NT:** Unsupported.
**Windows:** Use Windows 98 and later.
**Windows CE:** Unsupported.
**Header:** Declared in vidsvr.odl.
**Import Library:** Included as a resource in vid.ocx.

## BPCSuspend

[This is preliminary documentation and subject to change.]

The **BPCSuspend** object contains a single method, **DeviceRelease**, which you can use to suspend the video server and cause it to release all devices. For more information, see Suspending the Video Server.

### See Also

**BPCSuspended**

## BPCSuspend.DeviceRelease

[This is preliminary documentation and subject to change.]

The **DeviceRelease** method causes the video server to attempt to release all devices.

**Syntax**

```
object.DeviceRelease(Priority, pps)
```

**Parameters**

*object*
: Object expression that resolves to a **BPCSuspend** object.

*Priority*
: **Long** that indicates the priority of the suspension request.

*pps*
: Reference to a **BPCSuspended** object. If the video server successfully releases all video devices, this will contain a valid **BPCSuspended** object. Otherwise, this object contains **Nothing**.

**See Also**

**BPCSuspended**

## BPCSuspended

[This is preliminary documentation and subject to change.]

The **BPCSuspended** object indicates whether the video server has released all video devices.

This object is returned by the **BPCSuspend.DeviceRelease** method. If the method is sucessful and the Video server releases all video devices, the method returns a valid instance of **BPCSuspended**. The video server will remain suspended as long as this object instance exists.

To unsuspend the video server, release the **BPCSuspended** object by setting it to **Nothing**. The destruction of the **BPCSuspended** object notifies the video server that it can once again connect to devices. Otherwise, the video server will be re-enabled when the object variable goes out of scope or the application closes.

**See Also**

**BPCSuspend**

# Enhancement Video Control

[This is preliminary documentation and subject to change.]

The Enhancement Video control displays video within an *enhancement*. For example, in a multiframe enhancement layout you can use the Enhancement Video control to display the television show in one frame, while interactive content appears in the other frames. To display video using this control, you create an instance of the control in an enhancement page..

The Enhancement Video control is similar to the Video control in that both controls display video from an HTML page. However, two important differences exist between the Enhancement Video control and the Video control:

- The Enhancement Video control can be used only in HTML pages displayed by TV Viewer. In constrast, the Video control can be used in World Wide Web pages and stand-alone applications as well as in TV Viewer.
- The Enhancement Video control automatically handles the connection between itself and TV Viewer. Whereas, the Video control does not automatically handle such a connection.

If you are writing enhancement HTML pages that will only be displayed by TV Viewer, it is much simpler to use the Enhancement Video control than the Video control.

For more information on the Enhancement Video control, see the following topics:

- About the Enhancement Video Control
- Using the Enhancement Video Control
- Which Video Control Should I Use?

# About the Enhancement Video Control

[This is preliminary documentation and subject to change.]

The Enhancement Video control displays video in HTML enhancement pages hosted by TV Viewer. This control is implemented as the **Msepg4** class in the Msepg.ocx object library. The Enhancement Video control wraps the Video control, extending its functionality by automatically creating and handling a connection to TV Viewer when it is created.

**Note** The classes in Msepg.ocx other than **Msepg4** are reserved for use by Broadcast Architecture and are not supported for use by external applications.

The Enhancement Video control accepts an HTML parameter, INTENT, which indicates how the Enhancement Video control is used in the current Web page. If you use the Enhancement Video control to display video in an enhancement, INTENT should be set to the value ENHANCE_VIDEO. For more information, see Using the Enhancement Video Control.

## Using the Enhancement Video Control

[This is preliminary documentation and subject to change.]

To add video to an enhancement page using the Enhancement Video control, place an instance of the control in the HTML page in the location where you want the video to appear. The INTENT parameter of the control instance must be set to ENHANCE_VIDEO.

You can add an instance of the Enhancement Video control to your enhancement page by using HTML code such as the following:

```
<OBJECT
  ID=Vid
  CLASSID="clsid:a74e7f00-c3d2-11cf-8578-00805fe4809b"
  BORDER=0
  VSPACE=0
  HSPACE=0
  ALIGN=TOP
  HEIGHT=100%
  WIDTH=100%
>
    <PARAM NAME="INTENT" VALUE=ENHANCE_VIDEO>
</OBJECT>
```

To see an example of how TV Viewer uses the Enhancement Video control in its layout pages, view the source code for the Msvideo.htm and Video.htm layout files. Video.htm is a layout file used to display full-screen video. By default, these files are installed in the C:\Program Files\TV Viewer\Layouts\ directory.

**Note** The value FULL_SCREEN_VIDEO for the INTENT parameter is reserved for use by the default full-screen video layout of TV Viewer. This use is the only situation in which INTENT should be set to this value. Enhancement layout files must always set INTENT to ENHANCE_VIDEO.

## Which Video Control Should I Use?

[This is preliminary documentation and subject to change.]

To display video in an HTML file, you can use either the Video control or the Enhancement Video control.

The Video control is a general-purpose control that can be used in TV Viewer enhancements, World Wide Web pages, and stand-alone applications. In contrast, the Enhancement Video control is a TV Viewer–aware control that can only be used in TV Viewer enhancements. For example, you cannot use the Enhancement Video control to display video in an HTML file viewed by a Web browser.

However, because the Enhancement Video control implements functionality that interacts with TV Viewer, it is much simpler to use the Enhancement Video control to display video in enhancements than the Video control. To add video to an enhancement file using the Enhancement Video control, all you need do is create an instance of the control on an enhancement page. Video from the current channel is then automatically displayed.

In contrast, if you use the Video control instead of the Enhancement Video control you must write code or script to determine the enhanced show's channel number in the user's cable or satellite system and to set that channel as the control's input.

Using the Video control is more work than simply creating an instance of the Enhancement Video control, but the Video control provides the advantage that your application can programmatically control the tuning or video input. For example, a Video Valet application might keep a list of a user's preferred television line-up and automatically tune the Video control to the correct channel for each of the user's favorite shows.

To summarize:

- Use the Video control to display video in Web pages or stand-alone applications, or in enhancement files that require programmatic control over video input.
- Use the Enhancement Video control to display video in enhancement files that will only be displayed in TV Viewer and that do not require programmatic control over video input.

# DirectShow Filter Reference

[This is preliminary documentation and subject to change.]

This section documents features of the filters based on the Microsoft® DirectShow™ application programming interface (API) found on the broadcast client. Applications access these filters through the Microsoft® ActiveX™ control for video (the Video control). Applications use these filters to control playback and display of video and audio. In addition to the filters described here, broadcast clients use filters that are part of the Microsoft® Windows® 98 operating system. To locate a description of these filters, see Further General Information.

**Note** For reference purposes, each filter description following includes the name of the file containing that filter. However, you do not typically address a filter through that file name. Instead, you rely on the Video control to work with the Video Access server to set the connections for your application.

## Digital Crossbar Filter

[This is preliminary documentation and subject to change.]

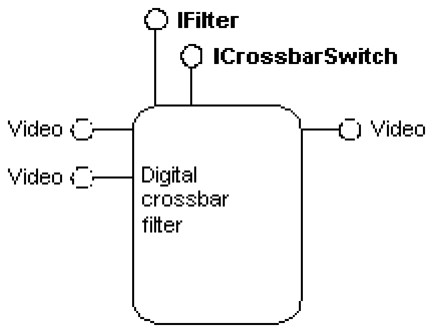
**File Name**

Tssxbar.ax

**Filter Type**

Utility filter

The following illustration shows a digital crossbar filter, with control interfaces and data pins.



**Summary**

The Digital Crossbar filter represents a digital video hardware multiplexer (MUX). Through this filter, a video source can be chosen and routed to the VGA device.

**Data Pins**

| **Name** | **Direction** | **Description** |
|---|---|---|
| Video | Input | Video type; UncompressedDigitalVideo subtype |
| Video | Output | Video type; UncompressedDigitalVideo subtype |

**Control Interfaces**

| **Name** | **Purpose** |
|---|---|
| **IFilter** | Provides run, stop, and pause capabilities |
| **ICrossbarSwitch** | Controls the switch |

# Announcement Listener

[This is preliminary documentation and subject to change.]

The Announcement Listener and its supporting components enable the *broadcast client* to receive announcements of upcoming data, and to programmatically filter incoming data at the network interface to select the data that the client computer should receive.

The following sections provide further information about the Announcement Listener:

# Announcements Overview

[This is preliminary documentation and subject to change.]

The broadcast of any data stream must be preceded by an announcement of the impending broadcast, describing the sender, address, time, contents, and other pertinent details. By examining these announcements, the broadcast client can select which data streams to keep and which data streams to ignore.

The Announcement Listener and its supporting components are designed to provide interfaces for capturing and processing data introduced by specific announcements.

The Announcement Listener enables installation of custom-defined filters to select announcements. Filters can be enabled and disabled individually. The Announcement Listener allows arbitrary data formats, both streaming and file-based.

The following sections provide an overview of announcements:

## Announcement Channels

[This is preliminary documentation and subject to change.]

To inform clients of the data broadcasts available on a network, broadcasts must be preceded by an announcement. An announcement is a small datagram containing information about the upcoming broadcast.

Announcements are sent on publicly known addresses. There may be more than one announcement address and port for any given network, and a broadcast client may be connected to more than one network on which announcements are sent.

While there is one well-known address on which general-interest announcements appear, it is expected that different data services residing on a single network may each have their own announcement channel for announcements that pertain only to those people that have subscribed to that service. On any network service, several vendors might each be offering various types of electronic data to their subscribers. Each vendor has its own announcement channel.

Segregating announcements into those that are of general interest and those that are of interest only to certain subscribers eases the load on users who do not subscribe to a particular service.

Regardless of the number of announcement channels, only one instance of the Announcement Listener need be running. The Announcement Listener is designed to listen on a very large number of Internet Protocol (IP) announcement addresses. (Each socket is listening on one IP announcement address.)

## Announcement Format

[This is preliminary documentation and subject to change.]

Session announcements are formatted according to Session Description Protocol (SDP) and its associated Session Announcement Protocol (SAP). These protocols are described in work-in-progress Internet draft documents produced by the Multiparty Multimedia Session Control (MMUSIC) working group of the Internet Engineering Task Force. As such, they may be updated, replaced, or made obsolete by other documents at any time.

To learn the current status of any Internet-Draft, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), nic.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (U.S. East Coast), or ftp.isi.edu (U.S. West Coast). The SDP and SAP draft proposals are described in the documents draft-ietf-mmusic-sdp-03.txt and draft-ietf-mmusic-sap-00.txt, respectively. For example, in the United States, the SDP document can be found at:

ftp://ftp.isi.edu/internet-drafts/draft-ietf-mmusic-sdp-03.txt

SAP is a header in binary format that precedes the SDP description, and is described in the SAP draft document. Both the SAP and SDP message announcements may be encrypted.

SDP is a textual protocol. It consists of series of lines of text, with each line beginning with a single letter followed by an equal sign, followed by a text line formatted appropriately, followed by a new line.

In the Microsoft® Windows® 98 operating system and version 4.0 of the Microsoft® Windows NT® operating system, the SDP protocol is wrapped inside a set of Component Object Model (COM) components and interfaces to send and receive announcements of multiparty multimedia conferences. For more information about SDP services in Windows, see Further Information on Data Services for the Client.

## Recommended Announcement Fields

[This is preliminary documentation and subject to change.]

Session Description Protocol (SDP) specifies a number of required and recommended fields (for more information, see Announcement Format).

In addition to these, the following attribute is available with Broadcast Architecture:

**•a=hidden**

Indicates that an announcement is not intended for human eyes — that is, it is intended for a programmatic filter.

## Sequence of Events

[This is preliminary documentation and subject to change.]

This overview describes how the various elements of the Announcement Listener work in time sequence.

First, an application installs a filter on the client computer. A filter is an Automation server, which can be either an in-process server implemented as a DLL (dynamic link library) file, or a local server implemented as an executable file.

The installation program causes the filter to register itself (using the **DLLRegisterServer** function, in the case of a DLL). The setup program then connects to the Announcement Listener, starting it if necessary, and invokes the **IFilterCollection::Add** method, supplying the programmatic identifier

(ProgID) of the registered filter. This method creates an instance of the filter (a *running filter*), and includes the instance in the list of filters recognized by the Announcement Listener.

When an announcement arrives on a socket associated with a well-known multicast announcement address, the Announcement Listener parses the announcement packet and puts the elements of the announcement in arrays of text lines, then wraps the arrays in an object (an instance of the **IBroadcastAnnouncement** method). The Listener checks that this announcement has not already been passed to the filters, and if not, it then hands that object to the filters. Each filter executes its own code to determine whether or not the announcement meets its criteria. The filter is called using the method **IBroadcastFilter::Match**.

If a filter returns a flag indicating it does not match, the Announcement Listener asks the next filter in turn, until all the filters have been asked.

If a filter returns with the match flag set to true, the Announcement Listener adds the announcement's unique identifier to a list, so that if the same announcement is rebroadcast later, the Announcement Listener does not ask the filters again.

If a filter returns a flag indicating that it has found a match, the filter also returns a second flag that can specify whether the Announcement Listener should schedule the launch of an application to receive the data at the expected time. In most cases, the filter makes this request. However, if the data is expected to arrive imminently, the filter can directly call an external process designed to receive the data. In this case, the filter sets the Schedule flag to "false," so that the Announcement Listener does not schedule a future event. Because the match flag is true, the Announcement Listener still treats the announcement as matched, and does not present the same announcement to the filters again.

If the Schedule flag is true, the Announcement Listener calls the filter's second method, **IBroadcastFilter::GetDisposition**. Through this method, the filter returns to the Announcement Listener the information it needs to schedule a future task to receive the data. This information includes what application to launch, the working directory, the command line, and how far in advance of receipt of the data the application should be launched.

In this case, the Announcement Listener provides the Task Scheduler with the announcement and the other information provided by **GetDisposition**. (For more information, see Scheduling Data Reception.)

After each announcement has been matched by a filter, or fails to be matched by any of the filters, the announcement is discarded.

At the appointed time, the Task Scheduler (a component of Windows 98) starts the receiver application, passing to it the appropriate Internet Protocol (IP) addresses and command line parameters.

The Announcement Listener also accepts announcements that cancel a previous announcement.

# Announcement Listener Components

[This is preliminary documentation and subject to change.]

The following sections describe components of the Announcement Listener:

## Announcement Listener System Service

[This is preliminary documentation and subject to change.]

The Announcement Listener system service comprises a number of components that operate together to capture, filter, and process broadcast data.

The primary Announcement Listener system service adheres to the **IDataListener** interface. It runs constantly unless explicitly disabled or halted by the user, receiving announcements from sockets on well-known Internet Protocol (IP) address and port pairs, passing those announcements to its installed filters, and scheduling the receipt of any announcements that a filter selects.

## Announcement Filters

[This is preliminary documentation and subject to change.]

Filters are used by the Announcement Listener to distinguish between announcements that are of interest to a user and those that are not. No individual wants to receive the data associated with every announcement, and no single computer has the capacity to do so.

The Announcement Listener maintains a list of filters. Each filter, when presented with an announcement, can register interest in the receipt of the data associated with the announcement. If a filter registers interest and requests the Announcement Listener to schedule future reception, it must also provide the name of an application to receive that data.

A filter is an Automation server that supports a number of interfaces. (For more information, see Creating an Announcement Filter.) Applications may add, remove, enable, or disable filters in the Announcement Listener through Automation, and users may do the same through the Announcement

Filter Manager. Filters have persistent states, and multiple instances of a given filter may be running simultaneously with each instance matching different criteria.

The Announcement Listener presents announcements sequentially to all the filters. However, once a filter registers interest in an announcement, that announcement and any rebroadcasts of it are not presented to any filter, to reduce the unnecessary overhead of calling filters that are not interested in the announcement. (The exception is when the **ListenAll** property of a filter is set to TRUE; for more information, see the **IBroadcastFilter:get_ListenAll** method.)

Future releases of Broadcast Architecture will ship with a simple, reprogrammable, generic filter class, which will suffice for many applications' filtering needs. It is expected that multiple instances of this generic filter will be run at the behest of different applications, each instance with its own criteria for matching announcements and its own rules for the disposition of the data received. An application that needs to perform simple filtering tasks that can be handled by the generic filter will be able to install a copy of that filter in the Announcement Listener and program that instance of the filter to match the announcements desired by the installing application.

## Receiver Applications

[This is preliminary documentation and subject to change.]

When an Announcement Listener filter registers interest in receiving a data stream, it must provide the name of an application for the actual capture (and possible storage) of that stream. The only defined interface between a filter and a receiver application is that of command line parameters, though obviously, if allied more tightly in their designs they can share richer communications.

The Announcement Listener spawns a receiver application before the broadcast of the desired data stream, and provide the receiver application with the Internet Protocol (IP) address and port of that stream as a command line parameter. The receiver application is then solely responsible for interpreting the format of the data stream, storing the data stream if desired, launching other applications, and taking any other appropriate actions. For more information, see Receiver Application Command Line.

Future releases of the Broadcast Architecture software will provide a generic receiver application for the transfer of files. This application would capture a file or files broadcast in the broadcast-architecture format, and optionally, either spawn a second application, passing the file names of the received file or files to it, or attempt to launch the files themselves (using the file associations for Microsoft® Windows® family of operating systems).

Thus, with the generic filter and generic receiver application, an application or individual is able to monitor announcements, receive files, and start arbitrary programs with those files. Only those applications that require more sophisticated filtration or receipt needs to implement their own filters and recipients.

## Receiver Application Command Line

[This is preliminary documentation and subject to change.]

A receiver application is an executable application that is intended to be launched by the Task Scheduler at the appropriate time to receive the information previously announced. You can use the generic receiver application or write one of your own. (For more information, see Receiver Applications.)

At the appropriate time, the Task Scheduler launches the receiver application. The Task Scheduler provides three command line arguments to the receiver application. These arguments are prepended to the *Parameters* string provided by the **IBroadcastFilter::GetDisposition** or **IDataListener::SubmitAnnouncement** method. These command-line arguments are:

- **/J** *job_name* — the task name in the **IJobScheduler** interface that launched this process. For example:

  ```
  /J BFTP_JOB_234
  ```

- **/IPD** *destination_address_and_port_of_multicast.* For example:

  ```
  /IPD 224.45.38.92:3256
  ```

- **/IPL** *local_address_of_network_card_receiving_multicast.* For example:

  ```
  /IPL 192.78.38.92
  ```

The destination address is the Internet Protocol (IP) address to which the data is broadcast. The local address is the IP address of the NIC that receives the data; this allows a distinction to be made between a satellite receiver card and any other type of network interface when more than one card is present.

For example, suppose the arguments supplied by a filter to the Announcement Listener by **IBroadcastFilter::GetDisposition** are as follows:

*WorkingDirectory* = **C:\Games\**
*Application* = **GameRecv.exe**
*Parameters* = **/o**
*AdvanceMinutes* = **2**

At the appointed time, Task Scheduler launches the specified receiver application with the following command line:

```
GameRecv /J "BFTP_JOB_234" /IPD 224.45.38.92:3256 / IPL 192.78.38.92 3A45F /o
```

In this case, GameRecv.exe is an application designed to receive multicast data on the specified address, and `/o` is an additional parameter that is being passed to the receiver application.

The receiver application can be the final receiver of the data (as in the example preceding), or it can be a simple shell that launches the final receiver of the data (for more information, see Using the Generic File Receiver Application).

## Scheduling Data Reception

[This is preliminary documentation and subject to change.]

The Announcement Listener must maintain a time-ordered queue of pending broadcasts and start the receiver applications sufficiently before their broadcasts so that they are loaded and ready when the data become available. If an announcement precedes its broadcast by a significant length of time, some client computers may be rebooted during that span. This means that the queue for broadcasts must be persistent.

Task Scheduler is a resource included with Windows 98. Task Scheduler enables the Announcement Listener to launch receiver applications at pre-specified times. The queue is stored on the hard disk, and thus persists across machine reboots. (Other components of the Broadcast Architecture also use the Task Scheduler for reminders of television shows.)

The Task Scheduler has a programmatic Component Object Model (COM) interface, as well as a simple user interface. The Announcement Listener uses only the programmatic interface.

# Creating an Announcement Filter

[This is preliminary documentation and subject to change.]

In order to respond to an incoming announcement, an appropriate filter must be created and installed on the client computer.

Announcement filters are instances of the class **IBroadcastFilter**. This interface has no intrinsic implementation by itself. Instead, implementations of particular filters are written for particular purposes. The generic filter is one such example.

**To write a filter**

- Register the filter as supporting component categories.

For information on adding the filter to the list of announcements used by the announcement, see Adding a Filter Programmatically.

## Registering Filters for Component Categories

[This is preliminary documentation and subject to change.]

Filters must register as supporting the component categories CATID_PersistsToStream and CATID_BpcFilter. CATID_PersistToStream is a standard category identifier in the Microsoft® Windows® family of operating systems, by which the filter advertises that it supports the interfaces necessary to save its data as a compound document stream.

The CATID_BpcFilter category identifier is unique to Broadcast Architecture. Components that register this class must adhere to the filter provided by the **IBroadcastFilter** interface.

The Announcement Listener registers these two categories upon startup.

Every filter must register the fact that it implements categories by calling **RegisterClassImplCategories**. Only filter classes that have been registered as supporting these two interfaces appear in the Filter Manager **Add Filter** dialog box. (The **Browse** dialog box causes a new file to register itself, so that it then appears in the list.)

Each instance of a filter must also register itself upon being created. The following code, written in the Microsoft® Visual C++® development system using Microsoft® Foundation Classes (MFC), illustrates how a filter must register the category IDs.

The **InitInstance** call performs the initialization of the filter. Because the filter is implemented in a dynamic linked library (that is, an in-process server), it calls the **COleObjectFactory::RegisterAll** method, which tells the system that it supports **IBroadcastFilter** and is a generic filter.

```
BOOL CFilterApp::InitInstance()
{
    HRESULT hr;
    // Register all OLE server (factories) as running. This enables
    // OLE libraries to create objects from other applications.
    COleObjectFactory::RegisterAll();
    // Create instance of the standard Component Categories Manager
    if (FAILED(hr = CoCreateInstance(CLSID_StdComponentCategoriesMgr,
                                     NULL,
                                     CLSCTX_INPROC_SERVER,
                                     IID_ICatRegister,
                                     (LPVOID *) &m_pCatRegister)))
    {
            AfxMessageBox(GetSystemMessage(hr,
                                     IDS_CANT_LOAD_COMP_MAN),
                                     MB_ICONSTOP|MB_OK);
        return FALSE;
    }
    // Register the component categories: specify the class ID of this
    // implementation of the filter, how many categories it supports,
    // and an array of the categories supported.
    else if (FAILED(hr = m_pCatRegister->
RegisterClassImplCategories(CLSID_GenericBroadcastFilter,
                            CGenericFilter::m_cImplCat,
                            CGenericFilter::m_rgImplCat)))
    // If registering class categories, display message box://
    {
            AfxMessageBox(GetSystemMessage(hr,
                                       IDS_CANT_ADD_CATEGORY),
                                       MB_ICONSTOP|MB_OK);
        return FALSE;
    }
    // Otherwise return TRUE, indicating successful initialization.//
    else
        return TRUE;
}
```

This completes the registering of the categories.

## Adding a Filter Programmatically

[This is preliminary documentation and subject to change.]

To add a new filter to the Announcement Filter Manager list of filters using programming code, first make sure the filter has already been registered as an Automation server. Then use the **IFilterCollection::Add** method.

(For development and testing, you may add a filter manually by using the Announcement Filter Manager.)

The following C++ code example illustrates the process of installing an announcement filter:

```
extern CLSID CLSID_MyFilter;

HRESULT hr;
IDataListener *pDataListener;
IFIlterCollection *pFilterCollection;
IBroadcastFilter *pNewFilter;
long lFiltIndex;
LPOLESTR lpszProgID    = NULL;
BSTR bstrProgID = NULL;

// Create an instance of the data listener:
hr = CoCreateInstance(CLSID_DataListener, NULL, CLSCTX_SERVER,
                        IID_IDataListener, (LPVOID *) &pDataListener);
if (FAILED(hr) || (NULL == pDataListener))
{
    return FALSE;
}
// Get pointer to data listener's filter collection:
else if (FAILED(hr = pDataListener->get_Filters(&pFilterCollection))
                        || (NULL == pFilterCollection))
{
    pDataListener->Release();
    return FALSE;
}
// Retrieve the ProgID for the filter:
hr = ProgIDFromCLSID(CLSID_MyFilter, &lpszProgID);
if (FAILED(hr))
{
    pDataListener->Release();
    pFilterCollection->Release();
    return FALSE;
}
bstrProgID = SysAllocString(lpszProgID);
CoTaskMemFree(lpszProgID); // Free memory allocated by ProgIDFromCLSID
if (NULL == bstrProgID)
{
    pDataListener->Release();
    pFilterCollection->Release();
    return FALSE;
}
// Create a running filter and add it to the filter collection:
hr = pFilterCollection->Add(bstrProgID, &pNewFilter, &lFiltIndex);
SysFreeString(bstrProgID);
if (FAILED(hr))
{
    pDataListener->Release();
    pFilterCollection->Release();
    return FALSE;
}
else
{
    pDataListener->Release();
    pFilterCollection->Release();
    pNewFilter->Release();
    return TRUE;
}
```

For more information, see the documentation for the **IFilterCollection::Add** method.

# Using a File Receiver Application

[This is preliminary documentation and subject to change.]

The following sections describe using a file receiver application:

- Using the Generic File Receiver Application
- Creating a Receiver Application
- Retrieving Announcement Attributes

## Using the Generic File Receiver Application

[This is preliminary documentation and subject to change.]

The Generic File Receiver is designed to capture a file or files that have been broadcast. The Generic File Receiver may also optionally spawn a second application, passing the file names of the received file or files to it, or attempt to launch the files themselves (using the file associations for the Microsoft® Windows® family of operating systems).

The Generic File Receiver expects to find the **/J**, **/IPD**, and **/IPL** options in the command line passed to it by the Task Scheduler, as described in Receiver Application Command Line.

In addition, the **/s** option may be used with the Generic File Receiver to spawn the received file. The **/s** string may be supplied in the *Parameters* string returned by **IBroadcastFilter::GetDisposition**. Without the **/s** option, the Generic File Receiver simply receives the file and saves it in the specified working directory.

For example, to launch the file received, the command line is:

```
GenRecv /J "BFTP_JOB_234" /IPD 224.45.38.92:3256 /IPL 192.78.38.92 /s
```

(The preceding example assumes that the Generic File Receiver has been named GenRecv.exe; in fact it has not yet been named.)

Any text on the command line of the Generic File Receiver beyond the **/J**, **/IPD**, **/IPL**, and **/s** options is treated as a new command line, with the name of the received file appended at the end of the command line arguments. For example, the command line:

```
GenRecv /J "BFTP_JOB_234" /IPD 224.45.38.92:3256 /IPL 192.78.38.92 Excel.exe
```

causes the Generic File Receiver to launch Microsoft® Excel with the file received. Assuming that the transmitted file was Sample.xls, this results in the following command line being launched:

```
Excel.exe Sample.xls
```

If a **/s** option appears with other arguments beyond the **/J**, **/IPD**, and **/IPL** on the command line, the **/s** are presented along with the other arguments as the new command line, rather than spawning the received file. For example:

```
GenRecv /J "BFTP_JOB_234" /IPD 224.45.38.92:3256 /IPL 192.78.38.92  Excel.exe /s
```

launches the following (assuming that the transmitted file was Sample.xls)

```
Excel.exe /s sample.xls
```

(In this case, **/s** is assumed to be a parameter intended for **Excel.exe**.)

If the Generic File Receiver runs successfully, it deletes its job from the Task Scheduler, even if there are later scheduled broadcasts (if there are no later scheduled broadcasts, the JOB_FLAG_DELETE_WHEN_DONE value causes the job to be deleted automatically, whether it succeeded or failed). If the Generic File Receiver fails and **IJob::GetNextRunTime** returns the value S_OK, then the job is left in the scheduler to run again.

## Creating a Receiver Application

[This is preliminary documentation and subject to change.]

If the Generic File Receiver application proves insufficient for the needs of your application, you can create a custom receiver application.

The application must be able to appropriate process the **/J**, **/IPD**, and **/IPL** options in the command line passed to it by the Task Scheduler, as described in Receiver Application Command Line.

Typically, a receiver application may ignore the **/J** option. The application must open a Windows Sockets (WinSock) socket using the address specified by **/IPD** and bind it to the address specified by **/IPL**. The application can then receive data over the broadcast connection and process it any way that is desired.

The receiving application may access various attributes that were included in the original announcement, as described in Retrieving Announcement Attributes.

## Retrieving Announcement Attributes

[This is preliminary documentation and subject to change.]

A receiving application, when spawned, may need to access any number of attributes from the original announcement. For example, when a stock ticker application is launched in response to announcement, it should display an appropriate title, depending on an attribute in the announcement (such as **a = market:NYSE**).

When the application is spawned, however, the original announcement is no longer available. Fortunately, the Task Scheduler stores the full text of the announcement, including all session attributes, in the user data for the task. The Task Scheduler supplies the name of the task that launched the receiving application in the command line passed to the receiving application. (For more information, see Receiver Application Command Line.)

The most convenient way to access the attributes in the task is to create a new broadcast announcement and fill it with the stored information from the task. Calling the **ITask::GetUserData** method in the Task Scheduler interface, and passing the result to the **IBroadcastAnnouncement::Load** method, fills an empty broadcast announcement with data from the Task Scheduler. The resulting announcement appears as though it had just been received by the Announcement Listener over an announcement channel.

The application is now able to access the attributes using the same announcement interface that announcement filters use.

## Specifying an Announcement Stream Source

[This is preliminary documentation and subject to change.]

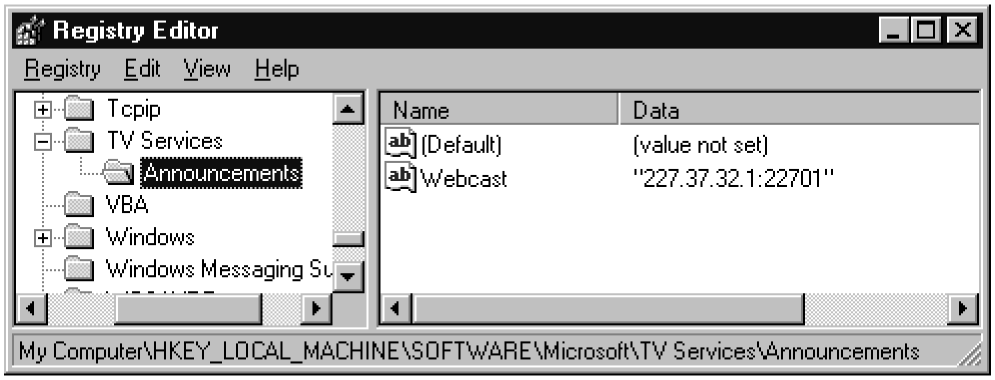
Announcements are broadcast on one or more streams. Each stream is associated with a particular data service, and is broadcast on a separate Internet Protocol (IP) address and port. The Announcement Listener can listen to more than one announcement stream at a time.

The names and addresses of the announcement streams are stored in the system registry, under the key:

**HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\TV Services\Announcements**

The Registry Editor (RegEdit) provides access to the names and addresses of announcement streams, and enables new announcement streams to be added.

In the **Announcements** key, each announcement stream is represented by a value. The Name of the value should reflect the name of the data service being announced. This name appears in dialog boxes and error messages. The Data field consists of the IP address and port of the announcement stream, expressed in the format *byte***.***byte***.***byte***.***byte***:***port*, as shown in the following illustration.



| Name | Data |
|---|---|
| (Default) | (value not set) |
| Webcast | "227.37.32.1:22701" |

My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\TV Services\Announcements

In the example shown, a data service called "Webcast" is being announced on IP address 227.37.32.1, port 22701.

Typically, the installation program for a data service programmatically installs a new value into the **Announcements** key, reflecting the name of the service and the address and port of its announcement stream.

# Announcement Listener Reference

[This is preliminary documentation and subject to change.]

The following sections provide reference material for the Announcement Listener:

- Filter Interfaces
- Announcement Listener Interfaces

## Filter Interfaces

[This is preliminary documentation and subject to change.]

Every broadcast filter installed in the Announcement Listener must be an Automation server supporting the following interfaces: **IBroadcastFilter** and **IPersistStream**. A filter supporting the interfaces **IGenericFilter** and its helper interfaces **IFields** and **IField** will be shipped with a future release of the Broadcast Architecture, but other filters are not required to support these interfaces.

Filters must have the normal registry entries for Microsoft® ActiveX™ components that share their functionality with other applications (local or in-process). In addition, they must register as supporting the component categories CATID_PersistsToStream and CATID_BpcFilter. For more information, see Registering Filters for Component Categories.

### Dual Interfaces

You can write your filter in any language that can implement dual interfaces. Choosing a language involves many considerations: familiarity with the language, tools support, run-time performance, threading models, code complexity, and the size of the compiled code. Languages and products that can be used to write Automation server components include Microsoft® Visual C++®, the Microsoft® Visual Basic® programming system, and Java.

### Announcement Listener Filter Interfaces

- **IBroadcastFilter**
- **IPersistStream**
- **IGenericBroadcastFilter**
- **IFilterFields**
- **IFilterField**

# IBroadcastFilter : IDispatch

[This is preliminary documentation and subject to change.]

The **IBroadcastFilter** interface has three purposes: to perform the actual filtration of broadcast announcements, to provide a user interface to a filter, and to provide methods and properties necessary for the enabling, disabling, or deletion of a filter.

Every filter must support Automation and the **IBroadcastFilter** interface. Each filter supplies the functionality for the methods in the table following, which are called by the Announcement Listener.

**Quick Info**

| **Called by:** | **Announcement Listener** |
|---|---|
| Interface definition: | Dual |

| **Method** | **Description** |
|---|---|
| **Delete** | The Announcement Listener invokes this method to notify the filters of an announcement deletion message. |
| **get_Enabled** | Indicates whether the Announcement Listener calls this filter. A filter may be disabled but still reside in the filter list. |
| **get_Hidden** | Returns a property that, if true, suppresses the display of the filter in the Filter Manager user interface. The filter still is accessible programmatically through the collection in the **IDataListener**. Hidden filters cannot be enabled, disabled, or removed by users. One use for a hidden filter is to listen for upgrades to the Broadcast Architecture software itself. |
| **get_LastMatched** | Property indicating the last time this filter selected an announcement. Used by the Filter Manager. |
| **get_ListenAll** | A property that, when true, causes all announcements to be presented to a filter, even if they have been matched previously by another filter. A filter with the **ListenAll** property set to TRUE receives all announcements, whether or not another filter has already registered to receive the associated |

| | |
|---|---|
| | broadcast data. This permits special filters to be created for monitoring or debugging. |
| **get_Name** | Returns the name of this instance of the filter. Used by the Filter Manager. |
| **get_NumberMatched** | The total number of announcements the filter has selected. Used by the Filter Manager. |
| **GetDisposition** | Returns to the Announcement Listener the information it needs to schedule a future task to receive the data, including what application to launch, the working directory, the command line, and how far in advance of receipt of the data should the application be launched. |
| **Match** | Checks whether the announcement describes a data transmission that should be received, and if so, whether the Announcement Listener should schedule a task to receive the data. Used by the Announcement Listener. |
| **put_Enabled** | Sets the **Enabled** property of the filter. |
| **ShowProperties** | The Announcement Listener invokes this method to request the filter to display its properties (if any) in a dialog box. |

## IBroadcastFilter::Delete

[This is preliminary documentation and subject to change.]

**IBroadcastFilter::Delete** is the method by which a filter is notified of an announcement deletion. Filters which schedule their data reception through the Announcement Listener need not respond.

However, filters which schedule their own data reception should authenticate the deletion packet and, assuming it is valid, remove any self-scheduled data receipts which were initiated by the announcement being deleted. Such filters must maintain a list of the origin lines (for example, **o=**) and source addresses from any announcements it receives. When a deletion is presented to the filter, the filter must ascertain that the deletion pertains to some announcement which it matched, and remove the scheduled receipt of the data from any queue the filter maintains.

Certain security verifications are required to ensure that an announcement deletion is legitimate. To locate more information on the relationship between an announcement and a deletion notice for that announcement, see Further Information on Data Services for the Client.

If a filter schedules tasks with the Announcement Listener, then the Announcement Listener provides all authentication.

```
HRESULT Delete(
  IBroadcastAnnouncement *Announcement  // pointer to the object
                                        // containing the deletion
                                        // announcement, as specified
                                        // by SAP.
);
```

### Return Values

Returns an **HRESULT** indicating success or failure.

## IBroadcastFilter::get_Enabled

[This is preliminary documentation and subject to change.]

This method retrieves the value of the **Enabled** property.

```
HRESULT get_Enabled(
  BOOL *pfEnabledRet  // pointer to flag indicating whether this
                      // filter is enabled.
);
```

### Return Values

Returns an **HRESULT** indicating success or failure.

### Remarks

The **Enabled** property specifies whether the Announcement Listener is to call this filter when a new announcement arrives. A filter may be disabled but still reside in the filter list.

## IBroadcastFilter::get_Hidden

[This is preliminary documentation and subject to change.]

The **Hidden** property, if true, suppresses the display of the filter in the Filter Manager user interface. The filter is still accessible programmatically through the collection in the **IDataListener**. Hidden filters cannot be enabled, disabled, or removed by users through the user interface, although they can be manipulated programmatically. One use for a hidden filter is to listen for upgrades to the Broadcast Architecture software itself.

```
HRESULT get_Hidden(
  BOOL *lpfReturn  // pointer to flag that indicates this filter is
                   // hidden.
);
```

**Return Values**

Returns an **HRESULT** indicating success or failure.

## IBroadcastFilter::get_LastMatched

[This is preliminary documentation and subject to change.]

This method returns the **LastMatched** property through a pointer.

The **LastMatched** property contains a date representing the last time this filter selected an announcement. Used by the Filter Manager.

```
HRESULT get_LastMatched(
  DATE *lpdateReturn  // pointer to the LastMatched flag.
);
```

**Return Values**

Returns an **HRESULT** indicating success or failure.

**See Also**

**IBroadcastFilter::get_NumberMatched**

## IBroadcastFilter::get_ListenAll

[This is preliminary documentation and subject to change.]

This method returns the **ListenAll** property through a pointer.

The **ListenAll** property, when true, causes all announcements to be presented to a filter, even if they have been matched previously by another filter. A filter with the **ListenAll** property set to TRUE receives all announcements, whether or not another filter has already registered to receive the associated broadcast data. This permits monitoring or debugging filters to be written.

```
HRESULT get_ListenAll(
  boolean *lpfReturn  pointer to the ListenAll flag.
);
```

### Return Values

Returns an **HRESULT** indicating success or failure.

## IBroadcastFilter::get_Name

[This is preliminary documentation and subject to change.]

The **IBroadcastFilter::get_Name** method returns the name of this instance of the filter. Used by the Filter Manager.

```
HRESULT Name(
  BSTR *lpbstrReturn  // out
);
```

### Parameters

*lpbstrReturn*
  Name of this instance of the filter.

### Return Values

Returns S_OK on success.

## IBroadcastFilter::get_NumberMatched

[This is preliminary documentation and subject to change.]

This method returns the **NumberMatched** property through a pointer.

The **NumberMatched** property contains the total number of announcements the filter has selected. Used by the Filter Manager.

```
HRESULT get_NumberMatched(
  long *lplReturn  // pointer to the number of matches.
);
```

### Return Values

Returns an **HRESULT** indicating success or failure.

### See Also

**IBroadcastFilter::get_LastMatched**

# IBroadcastFilter::GetDisposition

[This is preliminary documentation and subject to change.]

The **IBroadcastFilter::GetDisposition** method enables the filter to specify the information that the Announcement Listener needs to schedule a future task to receive the data, including what application to launch, the working directory, the command line, and how far in advance of receipt of the data should the application be launched.

```
HRESULT GetDisposition(
  BSTR * WorkingDirectory,  // out
  BSTR * Application,     // out
  BSTR * Parameters,      // out
  long * AdvanceMinutes   // out
);
```

### Parameters

*WorkingDirectory*
Working directory in which to launch the target application.

*Application*
File name of the application to be launched.

*Parameters*
String to be appended to the command line following the application name.

*AdvanceMinutes*
Number of minutes prior to the time the announced data arrives to launch the application.

### Return Values

Returns an **HRESULT** indicating success or failure.

### See Also

**IBroadcastFilter::Match**

# IBroadcastFilter::Match

[This is preliminary documentation and subject to change.]

The **IBroadcastFilter::Match** method checks whether the announcement describes a data transmission that should be received, and if so, whether the Announcement Listener should schedule a task to receive the data. If *Match* returns TRUE and the *Schedule* parameter is also TRUE, then the Announcement Listener creates a task in the Task Scheduler in the Microsoft® Windows® 98 operating system. This task receives the associated data. If *Match* returns TRUE and the *Schedule* parameter is FALSE, the announcement is treated as if it had been matched and scheduled, but no task is created.

```
HRESULT Match(
  IBroadcastAnnouncement *Announcement,   // in
  BOOL AlreadyMatched,                    // in
  BOOL *Schedule,                         // out
  BOOL *lpfReturn                         // out, retval
);
```

### Parameters

*Announcement*
: Pointer to the object containing the announcement.

*AlreadyMatched*
: Flag that indicates to a filter whether the announcement has already been matched by another filter. (This flag is of interest only to a filter designed for monitoring or debugging. For a filter that is not ListenAll, this parameter is always FALSE.)

*Schedule*
: Pointer to a flag used by the filter to request the Announcement Listener to schedule a task for future receipt of the data. Necessary scheduling information can be obtained by using the **IBroadcastFiler::GetDisposition** method.

*lpfReturn*
: Pointer to a flag used by the filter to indicate that the filter has found a match on the announcement data.

### Return Values

Returns an **HRESULT** indicating success or failure of the call.

### See Also

**IBroadcastFilter::GetDisposition**

## IBroadcastFilter::put_Enabled

[This is preliminary documentation and subject to change.]

This method stores the value of the **Enabled** property.

```
HRESULT put_Enabled(
  BOOL Enable  // flag indicating whether this filter is enabled.
);
```

#### Return Values

Returns an **HRESULT** indicating success or failure.

#### Remarks

The **Enabled** property specifies whether the Announcement Listener is to call this filter when a new announcement arrives. A filter may be disabled but still reside in the filter list.

Filters must shutdown any threads the filter is using when the Annoucement Listener calls **put_Enabled** with the **Enable** argument set to VARIANT_FALSE. This allows the Annoucment listener to properly shutdown.

## IBroadcastFilter::ShowProperties

[This is preliminary documentation and subject to change.]

**IBroadcastFilter::ShowProperties** is a method by which the Announcement Filter Manager may request a filter to display its properties with a dialog box. This method must be implemented by all filters, but the method need not necessarily display a dialog box.

Filters that contain properties that can be set by the user but cannot otherwise be edited can use this feature to display a dialog box through the Announcement Filter Manager. Hidden filters, and those filters with no editable properties, may implement this method with no action.

Filters that use this method to display a user interface should create a modal dialog box with the owner window set to the HWND parameter to the call. The filter *must* continue to service OLE requests, particularly calls to **Match** while the dialog box is displayed. This requirement implies that the call to **ShowPropertiesDialog** must return before the dialog box is displayed, because calling

**OleBlockServer** is not acceptable.

This may be accomplished either by using a modeless dialog box or by spawning another thread which creates a modal dialog box.

The following code must be included in all filters, at a minimum:

```
// In the class definition of your IBroadcastFilter implementation
STDMETHOD(ShowProperties)(OLE_HANDLE Window);

// In the implementation file.
STDMETHODIMP
CGenericFilter::XFilter::ShowProperties(OLE_HANDLE Window)
{
    METHOD_PROLOGUE(CGenericFilter, Filter)
    try
    {
        return E_NOTIMPL;
    }
    CATCH_DUAL_EXCEPT()
}
```

## IPersistStream

[This is preliminary documentation and subject to change.]

**IPersistStream** is a standard interface for saving and restoring the state of an object. Each filter, when created, may receive an **IStream** from which to initialize itself. The Announcement Listener uses this interface to request filters to save their states into a different **IStream** on shutdown, and to restore their states from that **IStream** on start-up.

At a minimum, a filter must save the number of announcements it has matched, and the date of the last announcement it matched. Any other state persistence is entirely at the option of the filter.

To locate more information on the **IPersistStream** interface, see Further Information on Data Services for the Client.

## IGenericBroadcastFilter

[This is preliminary documentation and subject to change.]

The generic filter will be provided in future releases of the Broadcast Architecture software. It will be programmable to match any announcement field, either as a regular or numeric expression. The disposition of the broadcast (receiving application, working directory, etc.) will also be settable.

This interface is currently in development, pending decisions on the announcement protocol.

The current version of the generic filter allows matching for any arbitrary line. In contrast, the ISDP interface contains specified properties including telephone, address, start time, stop time, and so on. With this interface, searching is not required; instead the generic filter will contain properties that roughly correspond to those of the ISDP interface, except that search strings, rather than values, are specified.

For example, the generic filter will include a URL match string which corresponds to the session URL string in the ISDP interface. The URL match string might contain http://www.microsoft.com/*.* to indicate all files at the Microsoft site. Each of these fixed properties will have a regular expression for matching.

For extensible attributes, the **a** field allows arbitrary fields. An arbitrary field can contain anything. The interfaces supports as many **a** lines as needed. The line may take either of the following forms:

**a**=*flag*
**a**=*attribute***:***value*

For example, the line **a=difficulty:friendly** specifies **friendly** as the value of the attribute **difficulty**.

## IFilterFields

[This is preliminary documentation and subject to change.]

This interface is the collection of fields in a generic filter that have been set to match announcement fields. With this collection, one can add or remove fields for matching, or iterate through the fields currently used by the filter.

Following is a preliminary description of the interface.

```
interface IFilterFields : IDispatch
{
// The Index parameter is numeric index or the name
[id(DISPID_VALUE)]
HRESULT Item([in] VARIANT Index, [out, retval] IFilterField ** lppifReturn);
[id(1), propget]
HRESULT Count([out, retval] long *lplReturn);
[id(2)]
HRESULT Add([in] BSTR Name);
[id(3)]
HRESULT Remove([in] BSTR Name);
[id(DISPID_NEWENUM), restricted]
HRESULT _NewEnum([out, retval] IUnknown **pUnkRetVal);
};
```

The **Item** method can take either a **BSTR** for the field name, or a numeric index.

## IFilterField

[This is preliminary documentation and subject to change.]

This is the heart of the generic filter; the interface through which the numeric or regular expression to match a field in an announcement can be set or retrieved.

Following is a preliminary description of the interface.

```
enum BPC_FieldType {AddressField, PatternField, NumericField, UndefinedField};

interface IFilterField : IDispatch
{
[id(DISPID_VALUE), propget] HRESULT MatchString([out, retval] BSTR *MatchString);
[id(DISPID_VALUE), propput] HRESULT MatchString([in] BSTR MatchString);
[id(1), propput] HRESULT FieldType([in] BPC_FieldType FieldType);
[id(3), propget] HRESULT Name([out, retval] BSTR* lpbstrReturn);
};
```

When added, a field has type *BPC_FieldType::UndefinedField*. The filter then only requires that the field be present in an announcement, without regard to its content. If a field is given a type of *BPC_FieldType:PatternField* or *BPC_FieldType:NumericField* then the *MatchString* is interpreted as a regular expression to be matched, or a numeric expression to be evaluated for truth or falsehood, respectively.

The numeric expression evaluator accepts the following operators: <, >, >=, <=, ==, !=, +, -, /, *, ), (, &&, ||, &, ~, |, ^, and !. Their meaning is similar to the same in the C language. Anywhere that a question mark appears in the expression, it is replaced by the numeric value of the field. All operations are carried out with double precision floating point numbers.

The regular expression evaluator uses accept the same patterns as the one in the Visual C++ version 4.1 IDE. This should be specified more completely in a future version of this document.

If a field type is set to *BPC_FieldType:AddressField*, then the value of the field is parsed as an Internet Protocol (IP) address. A variety of formats may be supported for this type.

## Announcement Listener Interfaces

[This is preliminary documentation and subject to change.]

The Announcement Listener supports Automation so that other applications can share its functionality. The Announcement Listener makes its methods available to other components through the following interfaces:

- **IDataListener**
- **IFilterCollection**
- **IBroadcastAnnouncement**

## IDataListener : IDispatch

[This is preliminary documentation and subject to change.]

This is the main interface of the Announcement Listener. The methods can enable or disable the program in its entirety, return the collection of filters, or submit a broadcast announcement to the scheduling queue without calling any filters.

| Method | Description |
|---|---|
| **get_AutoStart** | Retrieves the **AutoStart** property, which determines whether the Announcement Listener is included in the system startup. |
| **get_Filters** | A pointer to the collection of filters that are installed in the Announcement Listener. |
| **get_Running** | Determines whether the reception of announcements is enabled. |
| **IsBpcTask** | Returns TRUE if a task name from the Task Scheduler was created by the Announcement Listener, and FALSE if it was not. Task Scheduler is a component of Windows 98. |
| **put_AutoStart** | Stores the **AutoStart** property, which determines whether the Announcement Listener is included in the system startup. |
| **put_Running** | Enables or disables the reception of announcements from the announcement socket. |
| **SubmitAnnouncement** | Causes the Announcement Listener to unconditionally schedule the receipt of a broadcast, without the announcement being sent to the public announcement address. |

## IDataListener::get_AutoStart

[This is preliminary documentation and subject to change.]

```
HRESULT AutoStart(
  boolean *AutoStart  // pointer to the Autostart property
);
```

### Return Values

Returns an **HRESULT** indicating success or failure.

## IDataListener::get_Filters

[This is preliminary documentation and subject to change.]

The **IDataListener::get_Filters** method retrieves a pointer to the collection of filters that are installed in the Announcement Listener.

```
HRESULT Filters(
  IFilterCollection **ppFiltersRetVal  // Pointer to the collection
                                       // of filters
);
```

### Return Values

Returns an **HRESULT** indicating success or failure.

## IDataListener::get_Running

[This is preliminary documentation and subject to change.]

This method retrieves the value of the **Running** property.

```
HRESULT Running(
  BOOL *Run  // pointer to flag that indicates whether the
             // Announcement Listener is running.
);
```

### Return Values

Returns an **HRESULT** indicating success or failure.

## IDataListener::IsBpcTask

[This is preliminary documentation and subject to change.]

Given the name of a task in the Windows 98 Task Scheduler, this method indicates whether the task was submitted by the Announcement Listener. This method is not used by any other component of the Broadcast Architecture, but may be used by some other application.

```
HRESULT IsBpcTask(
  BSTR TaskName,       // in
  boolean *pfRetVal    // out, retval
);
```

### Parameters

*TaskName*
String containing a Task Scheduler task name, such as BFTP_JOB_234.

*pfRetVal*
Pointer to a flag that is TRUE only if the task was created by the Announcement Listener.

### Return Values

Returns an **HRESULT** indicating success or failure.

## IDataListener::put_AutoStart

[This is preliminary documentation and subject to change.]

```
HRESULT AutoStart(
  boolean AutoStart  // the Autostart property
);
```

### Parameters

*AutoStart*

Boolean value. Setting the *AutoStart* property to FALSE causes the Announcement Listener to remove itself from the system startup; setting the value for the property to TRUE adds the program to the system startup. Neither value affects the currently enabled or disabled state of the program.

**Return Values**

Returns an **HRESULT** indicating success or failure.

## IDataListener::put_Running

[This is preliminary documentation and subject to change.]

This method stores the value of the **Running** property.

```
HRESULT Running(
  BOOL Run  // flag that indicates whether the Announcement Listener
            // is running.
);
```

**Return Values**

Returns an **HRESULT** indicating success or failure.

## IDataListener::SubmitAnnouncement

[This is preliminary documentation and subject to change.]

The **IDataListener::SubmitAnnouncement** method allows an application to schedule the reception of a broadcast without an announcement being sent to the public announcement address. This method has two primary uses: to allow announcements to be distributed by other means, such as the Internet or in email, and to permit filters to store announcements for presentation to a user for approval.

For example, an application might consist of a filter to store announcements that may be of interest to a user, present them to the user at some later date, and register the announcement for receipt of its broadcast at that time. Announcements delivered by this mechanism are not presented to the filters, but scheduled unconditionally for the receipt of their broadcasts.

The input parameters to this method are equivalent to the output parameters of **IBroadcastFilter::Match** and **IBroadcastFilter::GetDisposition**.

In addition to the passed parameters, **SubmitAnnouncement**, must be able to access the local IP address by calling the **IBroadcastAnnouncment::get_LocalAddress** function. Therefore; applications that use **SubmitAnnouncement**, must call **IBroadcastAnnouncment::put_LocalAddress** prior to calling **SubmitAnnouncement**.

```
HRESULT SubmitAnnouncement(
  IBroadcastAnnouncement * Announcement,  // in
  BSTR WorkingDirectory,                  // in
  BSTR Application,                       // in
  BSTR Parameters,                        // in
  long AdvanceMinutes                     // in
);
```

### Parameters

*Announcement*
Pointer to the object containing the announcement.
*WorkingDirectory*
Working directory in which to launch the target application.
*Application*
File name of the application to be launched.
*Parameters*
String to be appended to the command line following the application name.
*AdvanceMinutes*
Number of minutes prior to the time the announced data arrives to launch the application.

### Return Values

Returns an **HRESULT** indicating success or failure.

### See Also

**IBroadcastFilter::GetDisposition**, **IBroadcastFilter::Match**

## IFilterCollection : IDispatch

[This is preliminary documentation and subject to change.]

The **IFilterCollection** interface represents the collection of filters that are installed in the Announcement Listener. This object has methods to add a filter, remove a filter, and iterate through all filters. It has the usual methods and properties for a collection: **Count**, **_NewEnum**, and **Item**.

| Method | Description |
|---|---|
| _NewEnum | Returns an **IenumVARIANT** of filters. |
| Add | Takes the Programmatic identifier of a filter as an argument and creates a new instance of the filter. |
| Count | Returns the number of currently installed filters in the collection. |
| Item | Returns a pointer to the indexed filter. |
| Remove | Removes the specified filter from the list. |

## IFilterCollection::_NewEnum

[This is preliminary documentation and subject to change.]

Returns an **IenumVARIANT** of filters.

```
HRESULT _NewEnum(
  IUnknown **ppUnk  // out, retval
);
```

### Parameters

*ppUnk*
Address of an interface pointer where this method returns the **IEnumVariant**.

### Return Values

Returns an **HRESULT** indicating success or failure.

### See Also

**IFilterCollection::Count**, **IFilterCollection::Item**

## IFilterCollection::Add

[This is preliminary documentation and subject to change.]

The **IFilterCollection::Add** method is the means by which the Announcement Listener creates a new instance of a given filter.

**Add** takes the programmatic identifier (ProgID) of a filter as an argument (for example, **BPC.GenericFilter** for the generic filter). Using the programmatic identifier (ProgID), **Add** then looks up the corresponding class identifier (CLSID), and calls **CoCreateInstance**, which makes a running instance of the object. If the filter is an in-process server, it runs in the process of the Announcement Listener.

```
HRESULT Add(
  BSTR ProgID,                     // in
  IBroadcastFilter ** NewFilter,   // out
  long *plRetVal                   // out, retval
);
```

### Parameters

*ProgID*
    Programmatic identifier of the filter to be added.
*New Filter*
    Pointer to the running filter than has been created as a result of the **Add** method.
*plRetVal*
    Index into the collection. This index can later be passed back as an argument to **Item** or **Remove**.

### Return Values

Returns an **HRESULT** indicating success or failure.

### See Also

**IFilterCollection::Remove**

## IFilterCollection::Count

[This is preliminary documentation and subject to change.]

The **Count** property returns the number of currently installed filters in the collection.

```
HRESULT Count(
  long *plRetVal  // out, retval
);
```

### Parameters

*plRetVal*
Pointer to the number of filters.

### Return Values

Returns an **HRESULT** indicating success or failure.

### See Also

**IFilterCollection::Item**, **IFilterCollection::_NewEnum**

## IFilterCollection::Item

[This is preliminary documentation and subject to change.]

This method returns a pointer to the indexed filter.

```
HRESULT Item(
  long Index,                          // in
  IBroadcastFilter ** ppFilterRetVal  // out, retval
);
```

### Parameters

*Index*
Index of the filter from which to retrieve information.
*ppFilterRetVal*
Pointer to the indexed filter.

### Return Values

Returns an **HRESULT** indicating success or failure.

### See Also

**IFilterCollection::Count**, **IFilterCollection::_NewEnum**

## IFilterCollection::Remove

[This is preliminary documentation and subject to change.]

The **Remove** method takes an index into the collection, and removes the specified filter from the list.

```
HRESULT Remove(
  long Index    // in
);
```

### Parameters

*Index*
Index into the collection specifying the filter to remove.

### Return Values

Returns an **HRESULT** indicating success or failure.

### See Also

**IFilterCollection::Add**

## IBroadcastAnnouncement

[This is preliminary documentation and subject to change.]

An **IBroadcastAnnouncement** object encapsulates the Session Announcement Protocol (SAP) and Session Description Protocol (SDP) sections of an announcement. This interface wraps the Component Object Model (COM) object **ISDP**, which in turn encapsulates the SDP COM object and some SAP fields.

This interface is currently in development, pending decisions on the announcement protocol.

## Announcement Filter Manager

[This is preliminary documentation and subject to change.]

The Announcement Filter Manager (Annui.exe) serves as a user interface to the Announcement Listener.

This tool provides developers of broadcast client applications with an overview of the state of running announcement filters. Using Announcement Filter Manager, you can install a filter manually, without using an installation setup routine, and manually remove, enable and disable filters. Usually, installing and enabling a filter is done programmatically, rather than through Announcement Filter Manager.

The Announcement Filter Manager interface is analogous to the Microsoft® Windows NT® Task Manager in that it is useful for programmers or advanced users, but typically it is not available from a shortcut or menu item.

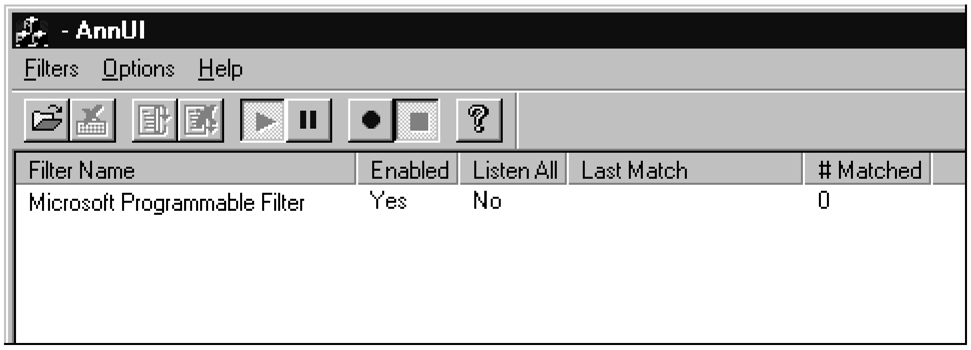
The following topics provide more information about Announcement Filter Manager:

- Announcement Filter Manager Main Window
- Announcement Filter Manager Toolbar
- Adding a Filter with the Filter Manager

## Announcement Filter Manager Main Window

[This is preliminary documentation and subject to change.]

The main window of the Announcement Filter Manager, shown in the following illustration, displays the names of and details about the installed announcement filters in a list. The main window also includes menus with commands that add, remove, enable, and disable filters, a toolbar that duplicates some menu functions, and a status bar.



| Filter Name | Enabled | Listen All | Last Match | # Matched |
|---|---|---|---|---|
| Microsoft Programmable Filter | Yes | No | | 0 |

The columns in the filter list are:

- **Filter Name** — the full name of a particular filter.
- **Enabled** — whether or not this filter is enabled.
- **Listen All** — whether this filter is configured to be notified of all incoming announcements, even when another filter has registered interest in those announcements. Filters that have this property enabled can be used for monitoring or debugging announcements.

- **Last Match** — the date and time when this filter last matched an announcement (if ever).
- **# Matched** — how many times this filter has registered a match.

To select a filter in the list, click it. When a filter is selected, options for manipulating that filter become available on the toolbar.

## Announcement Filter Manager Toolbar

[This is preliminary documentation and subject to change.]

As shown in the illustration in Announcement Filter Manager Main Window, the Announcement Filter Manager features a toolbar with the following buttons:

- **Add Filter** — adds a new filter to Announcement Listener's filter list. For more information, see Adding a Filter with the Filter Manager.
- **Remove Filter** — removes the selected filter from Announcement Listener's filter list. Clicking this button does not delete any information about the filter from the registry, or delete any files.
- **Enable Filter** — sets a filter property so that the Announcement Listener calls this filter when a new announcement arrives. You cannot click this button unless the selected filter is disabled.
- **Disable Filter** — sets a filter property so that the Announcement Listener does not call this filter when a new announcement arrives. You cannot click this button unless the selected filter is enabled.
- **Start Announcement Listener** — starts reception of data announcements and announcement processing. You can only click this button when the Announcement Listener is stopped.
- **Stop Announcement Listener** — stops all reception of data announcements. Clicking this button disables all announcement processing until the computer is restarted, but it does not stop the receipt of previously scheduled broadcasts. You can click this button only when the Announcement Listener is running.
- **Auto Start** — adds the Announcement Listener to the startup (**RunServices**) section of the registry, so that the Announcement Listener is launched when the machine is restarted. (Usually a broadcast client is configured to launch Announcement Listener at startup.)
- **No Auto Start** — removes the Announcement Listener from the startup section of the registry.
- **About Dialog Box** — displays the **About Announcement Listener** dialog box.

## Adding a Filter with the Filter Manager

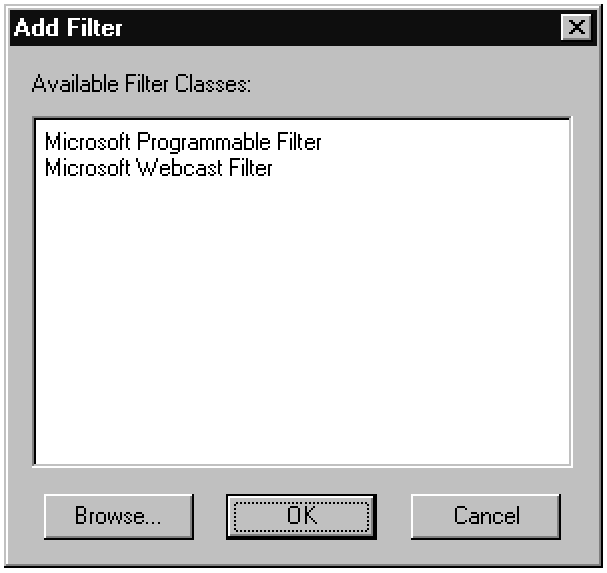
[This is preliminary documentation and subject to change.]

The following procedures describe how you can use Announcement Filter Manager to create new instances of running filters and to add these running filters to the list of running filters maintained by the Announcement Listener.

**To add a new announcement filter to the list of running filters**

1. Click the **Add Filter** button, or select **Add Filter** from the **File** menu.

   Either of these options displays the **Add Filter** dialog box, shown following.



Microsoft Programmable Filter
Microsoft Webcast Filter

The **Available Filter Classes** list includes all filter classes that have been entered in the computer's registry with a category identifier of CATID_BpcFilter.

2. Click the appropriate filter class in the **Available Filter Classes** list.
3. Click **OK**.

**To add a new filter from a filter class that is not yet registered**

1. Click **Browse**.

   A **Browse** dialog box appears.

2. Click the name of the filter file you want to register and add. The file may be an executable (.exe) or dynamic-link library (.dll) file.
3. Click **OK** to register the selected filter class.

   Announcement Filter Manager enters information about the type library for the filter class in the registry. The new class appears in the list of available filter classes.

4. To create a running filter from this new class, click **OK**.

Note that the **Add Filter** dialog box lists all possible classes of filters, as opposed to all filters currently running. A filter object is an instance of a filter class, and more than one instance of a filter class might be running concurrently. For example, the generic filter class is listed only once in the **Add Filter** dialog box, but many instances of the generic filter might be running. To see a list of all currently running filters, check the Announcement Filter Manager main window.

# Video Enhancements

[This is preliminary documentation and subject to change.]

The term *enhanced video* refers to the combination of a video program and multimedia elements such as hypertext links, graphics, text frames, sounds, and animations.

This section is directed to those who plan to create or display enhanced video using Broadcast Architecture.

The sections listed following provide an overview the enhancement components and their interactions, instructions on using the enhancement components to create and receive enhancement streams, and reference information for the enhancement components.

- About Enhancements explains what enhancements are and describes the various components of the enhancement system.
- Using Enhancements explains how to create, transmit, and display enhanced shows.
- Enhancement Reference contains reference material relating to enhancements and the objects used to create and view them.

# About Enhancements

[This is preliminary documentation and subject to change.]

The following topics describe the components and procedures used with enhancements:

- Enhanced Shows explains what an enhanced show is and lists some of the features you can implement.
- Enhancement Stream discusses the script used to synchronize enhancements with video.
- Stream Compiler Objects describes objects you can use to build enhancement stream editing tools.
- Enhancement Sender Object describes the object used to transmit enhancement events and files across a multicast router or LAN.
- Enhancement Client Architecture illustrates the reception and flow of enhancement data on the Broadcast Architecture client.
- Enhancement Client Controls, describes the controls used to receive and display enhanced video on the client.
- Location of Enhancement Files on the Client explains where enhancement files are stored on the client computer.
- Enhancement Filter describes the enhancement filter and the programmable objects that interact with it.

## Enhanced Shows

[This is preliminary documentation and subject to change.]

An enhanced show is one where the video is accompanied by interactive Web-style content. This content can take many forms: simple text, rotating advertisements, links to Internet content, even interactive chat controls that enable viewers to discuss the show with other viewers.

Because Broadcast Architecture uses HTML and Web technology as the basis for enhancements, any type of content that can be delivered over the Web can also be used to build an enhanced show.

An enhancement can use functionality such as rotating content, hyperlinks, Applets, and Microsoft® ActiveX™ controls. Enhancements can even interact with TV Viewer by scripting a .dll that implements an **ITVControl** sink.

## Enhancement Stream

[This is preliminary documentation and subject to change.]

The enhancement stream is a script that synchronizes enhancement events with specific times during the enhanced show. An enhancement event changes the enhancement state and usually requires a response by the client.

For example, an enhancement announcement sends data about a future enhanced show to the client. Controls on the client handle this event and update the show listing in the Guide database.

There are three types of enhancement events currently defined:

- Enhancement announcements, which transmit information about an upcoming enhanced show.
- FTS downloads, which use FTS protocol to download enhancement files and dependencies.
- Triggers, which command the client to perform pre-defined actions. For example, one type of trigger causes TV Viewer to automatically display a new enhancement page.

When an enhancement stream is saved to a file, the enhancement events are stored as stream compiler statements. To locate more information about the stream compiler syntax, see Further Information on Data Services for the Client.

### Enhancement Announcements

[This is preliminary documentation and subject to change.]

Enhancement announcements inform the client that a show is enhanced. They provide details about the enhancement such as the enhancement identifier, the show that it enhances, and the starting page.

When the enhancement filter receives an enhancement announcement, it stores the enhancement information in the Guide database. The presence of enhancement data in a show listing indicates that the show is enhanced. Each time that a user tunes TV Viewer to a new channel, TV Viewer checks the Guide database, to see if the new show is enhanced. If it is, and the user has enhancements enabled, TV Viewer will display the show as enhanced.

There are two types of enhancement announcements, announcements for future enhanced shows and announcements for enhanced shows that are currently being broadcast.

For more information, see the following topics:

- Future Enhancement Announcements
- Current Enhancement Announcements

## FTS Downloads

[This is preliminary documentation and subject to change.]

The enhancement files and their dependencies are transmitted to the client computers using File Transfer Service (FTS). FTS is a component of Microsoft® NetShow™ server that can send files using a *multicast* transfer mechanism that includes *forward error correction*. NetShow is a component of Microsoft® Site Server.

For more information about FTS, see Further Information on Data Services for the Client.

## Triggers

[This is preliminary documentation and subject to change.]

Triggers are notifications that are sent to the client at specific times during the show. They instigate or trigger an action on the client. For example, you can use triggers to automatically change the displayed content, as in an advertisement rotator.

There are fours types of triggers currently supported by Broadcast Architecture:

- FTSData, causes the client to tune to a specified IP address and port to receive File Transfer Service (FTS) data

- NavBase, displays the specified HTML file as the top frame. In other words, the file is displayed as if the user had clicked on an HREF hyperlink that used the parameter TARGET = TOP.
- NavFrame, displays a new page in a particular HTML frame.
- UserTrigger, sends a user trigger event. The functionality of a user trigger is defined by the content provider. The content provider should implement scripts in the enhancement page to handle custom triggers they define.

Triggers are received by the enhancement control, **EnhCtrl** which handles Broadcast Architecture-defined triggers and passes user-defined triggers to scripts and, and if applicable, the user trigger control, **EnhUser**.

### Dependencies

[This is preliminary documentation and subject to change.]

Enhancement pages, like Web pages, are typically composed of several files. For example an enhancement file, MyEnh.htm, can contain .gif and .jpeg graphic images, ActiveX controls, FutureSplash animations, and other such files that must be downloaded for the enhancement page to display properly. These additional files are MyEnh.htm's dependencies.

You must ensure that the enhancement's dependency files are stored on the client computers before the enhancement is displayed. There are several strategies for handling dependencies:

- Allocate broadcast bandwidth to download the dependencies.
- Use another data transmission method, such as webcasting, to get the files onto the user's computer.
- Have the user explicitly download and/or install the dependency files, either from a Web site or portable media such as a CD-ROM.

Note that the first method, while the most elegant, requires careful budgeting of the bandwidth. You must also handle the situation where a user tunes to a television show in the middle of a broadcast, possibly missing transmission of some or all of the enhancement files. Typically, this problem is handled by repeatedly transmitting files during the show.

The enhancement stream editor provides functionality to assist you in budgeting bandwidth for enhancements and their dependencies. To locate more information on the enhancement stream editor, see Further Information on Data Services for the Client.

## Stream Compiler Objects

[This is preliminary documentation and subject to change.]

The stream compiler objects are a set of COM objects that provide functionality to load, edit, and save enhancement streams. Using these objects, you can create enhancement stream authoring tools. For example, you can create applications such as an enhancement stream editor to compile enhancement streams or a stream player to view and test the enhanced show.

**Note** The stream compiler object library, Stream.dll, is not part of the software supporting the Broadcast Architecture Programmer's Reference. To locate this library, see Further Information on Data Services for the Client.

The stream compiler object library contains the following objects:

- **Event**, an object that wraps information about a particular enhancement event, such as an announcement, FTS data transfer, or trigger.
- **Events**, a collection of **Event** objects. Using this object, you can manipulate the entire stream of events, rearrange events, search for a specific event, set default values, load or save the stream to a file, and so on.

Another object that is useful if you are writing enhancement tools is the enhancement sender object, **ipsend**. This object connects to the broadcast medium and can transmit all three types of enhancement events: enhancement announcements, FTS downloads, and triggers.

## Enhancement Sender Object

[This is preliminary documentation and subject to change.]

The enhancement sender object, **ipsend**, is an object that you can use to transmit the enhancement stream over a broadcast medium such as a multicast router or LAN. The **ipsend** object supports three separate IP connections, one each for: enhancement announcements, triggers, and FTS downloads.

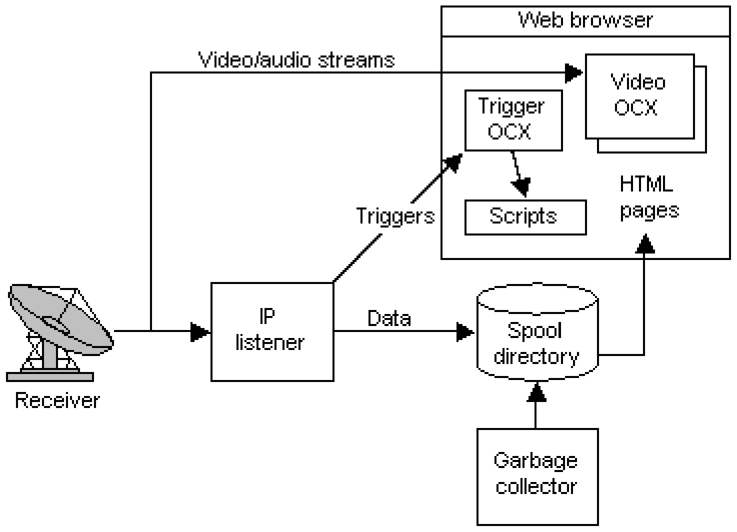
For more information, see Broadcasting Enhancements.

**Note** The stream compiler object library containing **ipsend**, Stream.dll, is not part of the software supporting the Broadcast Architecture Programmer's Reference. To locate this library, see Further Information on Data Services for the Client.

## Enhancement Client Architecture

[This is preliminary documentation and subject to change.]

On the broadcast client (the home viewer's personal computer), data is received as Internet data. From this incoming stream, the video and audio data are extracted and sent to the video control, which resides in the page displayed by TV Viewer or a Web browser. At the same time, the enhancements and triggers are received by the enhancement client controls. These controls are ActiveX controls that respond to enhancement stream events. The following illustration shows this process.



You can extend the functionality of the client enhancement controls by scripting applications using Microsoft® Visual Basic® Scripting Edition (VBScript) or Microsoft® JScript™ development software that run in the client environment. Typically this is done by implementing scripts that handle trigger events. For more information, see Handling Triggers.

## Enhancement Client Controls

[This is preliminary documentation and subject to change.]

The enhancement client controls are ActiveX controls that can be embedded in enhancement pages. These controls receive and respond to triggers, and send trigger events.

The main client-side control is the enhancement control, **EnhCtrl**. This control monitors the enhancement trigger stream and responds to triggers that the content provider broadcasts. For example, if the content provider sends an FTSData trigger, signaling an impending FTS broadcast, **EnhCtrl** creates an internal object to tune to the specified IP address and port and receive the data.

If **EnhCtrl** receives a trigger that it does not recognize, in other words a user trigger, it sends a trigger event that can be handled by scripts. There is also a user trigger control, **EnhUser**, that sends user trigger events. You can create **EnhUser** on pages where you wish to script user-trigger event handlers. This object is useful because unlike **EnhCtrl**, which can only be created once in a given enhancement, you can create multiple instances of **EnhUser**.

For more information, see the following topics:

- Enhancement Control

- User Trigger Control

### Enhancement Control

[This is preliminary documentation and subject to change.]

The enhancement control, **EnhCtrl**, is a control that you can create in enhancement pages to receive and handle broadcast triggers.

When TV Viewer displays an enhanced show, it tunes **EnhCtrl** to the IP address and port specified in the enhancement data stored in the Guide database. (This is the enhancement data that was stored during a prior enhancement announcement.) **EnhCtrl** monitors this IP stream for triggers. When **EnhCtrl** receives a trigger, it handles Broadcast Architecture triggers, and sends an event for user triggers. For example, if the content provider sends a NavBase trigger, **EnhCtrl** will cause TV Viewer to display the enhancement's base page.

Note that you should only have one instance of **EnhCtrl** created at a time. This is because multiple instances can result in redundant file reception and navigation. For example, if you create three instances of **EnhCtrl**, one on each frame of an enhancement, each instance of **EnhCtrl** will perform the action commanded by incoming triggers. A trigger to display a new page in the left-hand frame will cause all three instances to load the new page into the frame. This limitation does not apply to **EnhUser**, you can create as many instances of each as you want.

For more information, see Receiving Triggers.

### User Trigger Control

[This is preliminary documentation and subject to change.]

The user trigger control, **EnhUser**, is a control that you can add to enhancement pages to generate trigger events for user-defined triggers. It sends a UserTrigger event each time the client receives a user trigger.

You can script event-handling routines on an enhancement page containing a **EnhUser** object. This enables you, the content provider, to define and implement handler for custom triggers.

For more information, see Handling User Triggers.

## Location of Enhancement Files on the Client

[This is preliminary documentation and subject to change.]

Enhancement files may be located in either of two directory structures, the spool folder or the base page folder.

### Spool Folder

The *spool folder* serves as the location where all enhancement files are received during a broadcast. The spool folder is created during the installation of Broadcast Architecture and can be found through the system registry. For each enhanced show that is received, the client control creates a subfolder, called an *episode folder,* within the spool folder, and gives the episode folder a unique name. The client control then saves the enhancement files for the specific episode in the episode folder.

The episode folder can be further divided into subfolders. File names are specified relative to the episode folder. For example, if the client control receives the transmitted file Images\Logo.jpg, then the client control creates a folder called Images within the episode folder, if it does not already exist, and writes the file Logo.jpg to that folder.

### Base Page Folder

The *base page folder* is the folder that contains the HTML page that resides at the top level of pages loaded into the Web browser. The client trigger control examines the Uniform Resource Locator (URL) of the top-level page, extracts the path information from the URL, and considers that path to indicate the base page folder. If the top-level page has been broadcast along with other enhancements, then the base page folder is the same as the episode folder. However, the top-level page can also reside in some other directory on the broadcast client.

For example, the top-level page might be a file that was installed initially along with the Broadcast Architecture, such as a generic frame structure. These generic pages reside in the Program Files\TVViewer\Layouts folder. Alternately, the top-level page may have been downloaded by the viewer from a Web site prior to the start of the broadcast. In this case, the base page folder is the folder where the viewer saved the top-level page.

When the enhancement client control receives a trigger to navigate to a file, the control first looks for that file in the episode folder, or a subfolder if specified. If the file is not found there, the control next looks in the base page folder, if different than the episode folder. An enhanced show may use a combination of files from either or both the episode folder and the base page folder.

If the file is not found in either location, the client control displays an error message when in authoring mode and does nothing in normal mode.

For more information, see the Remarks section in the **EnhCtrl** topic.

## Enhancement Filter

[This is preliminary documentation and subject to change.]

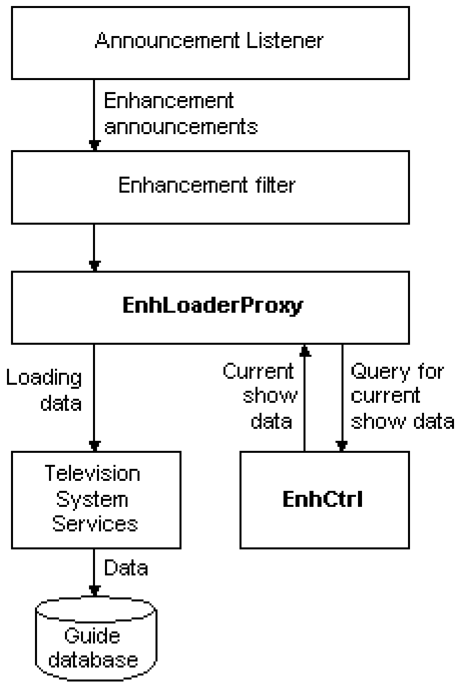
The enhancement filter is an Announcement Listener filter that processes enhancement announcements. Enhancement announcements contain data about an enhanced show, such as the filename of the base page, the name of the enhancement, and the show reference of the enhanced show.

When the enhancement filter receives an announcement, it calls the enhancement loader proxy object to load the enhancement data into the Guide database. Applications such as TV Viewer check for this information to determine whether a show is enhanced and locate its base page.

**Note** The data passed to the enhancement filter through announcements only specifies the parameters of the enhancement. It does not contain content such as enhancement HTML files. These files are broadcast to the user's computer by means such as File Transfer Service (FTS) transmission, Internet channel broadcasting, and so on.

There are two types of enhancement announcements, announcements for future enhanced shows and announcements for enhanced shows that are currently being broadcast. For more information, see Future Enhancement Announcements and Current Enhancement Announcements.

The following diagram illustrates the flow of enhancement data from the announcement stream to the Guide database.



There should be only one instance of the enhancement filter running on the computer at any time. Additional instances after the first do not receive enhancement announcements and slow performance. This is because the Announcement Listener compares the type of each incoming announcement against all registered filters. If there are multiple enhancement filters, this increases the number of comparisons the Announcement Listener must perform.

## Future Enhancement Announcements

[This is preliminary documentation and subject to change.]

Future enhancement announcements contain information about the enhancements of an upcoming broadcast. These announcements fully specify all data, including multicast address, port, ending time, base page, and show reference. The enhancement filter simply loads this data into the Guide database by calling the enhancement loader proxy object.

## Current Enhancement Announcements

[This is preliminary documentation and subject to change.]

Announcements for current broadcasts are used to match an enhancement to an episode that the viewer has tuned to in the middle of the broadcast. In this case, the client has not received the initial enhancement trigger at the beginning of the show. Current enhancement announcements provide a method for applications such as TV Viewer to start an enhancement with an episode in midstream.

Current announcements can also be used when the show time is not known. An example of this is the case where video and its enhancement stream (which includes the announcements) is recorded to tape for later broadcast. Using current announcements ensures the announcements will work no matter when the show is broadcast.

Current announcements expire immediately; in other words, their expiration date is the same as their transmission date. This ensures the Announcement Listener re-receives each broadcast of the current enhancement announcement. Otherwise, the default behavior of the Announcement Listener is to ignore duplicate announcements.

Current enhancement announcements are different from future enhancements in that they specify the ending time as the show length plus five minutes instead of as a fixed time in the show reference. For example, a future enhancement announcement might specify the ending time as 1/26/98 12:00:00, whereas a current enhancement announcement might specify it as 35 minutes.

When the enhancement filter receives a current enhancement announcement, it calculates the ending time and passes the connection information to the enhancement loader proxy object. If the show reference or the base page of the enhancement, the preload URL, is not specified in the announcement, the enhancement loader proxy gets the show reference and preload URL values of the current show from the enhancement control. When the enhancement information is complete, the enhancement loader proxy loads the enhancement data into the Guide database.

## Enhancement Loader Proxy

[This is preliminary documentation and subject to change.]

The enhancement loader proxy, **EnhLoaderProxy**, is an object that loads data from an enhancement announcement into the Guide database. The enhancement filter creates and calls an instance of this object to load the data from enhancement announcements into the Guide database. This is the same data that TV Viewer checks to determine whether a show is enhanced.

You can use **EnhLoaderProxy** in applications where you need to explicitly load enhancement data into the Guide database. For example, if you created a Web site where users can download enhancement files, your application could call **EnhLoaderProxy** to update the user's Guide database with the new enhancement data.

## Using Enhancements

[This is preliminary documentation and subject to change.]

By adding enhancements to video, you can make a presentation richer in content and more interactive. This section explains the technical details required to create, transmit, and receive an enhanced show. To locate information on design issues, see Further Information on Data Services for the Client.

The following topics describe the processes involved in creating an enhanced show:

- Creating Enhancement Files, explains how to create enhancement pages.
- Creating an Enhancement Stream describes how to create a script that synchronizes the download and display of your enhancements with the enhanced show.
- Flattening an Enhancement Stream describes flattened and unflattened stream syntax and explains how to flatten a enhancement stream.
- Broadcasting Enhancements, describes the process of broadcasting enhancements to clients.
- Receiving Enhancements on the Client describes how to use the client controls in enhancement pages to enable users to receive the various types of enhancement data.
- Displaying Enhancement Pages in TV Viewer, describes how TV Viewer recognizes an enhanced episode and what you must do to register your enhancement with TV Viewer.
- Handling Triggers, explains how to use the enhancement client controls to handle trigger events.
- Controlling Navigation in Enhancements, describes how to use navigation triggers and Microsoft® Visual Basic® Scripting Edition (VBScript) subroutines to display synchronized enhancements in the appropriate frames. This section also explains how to implement viewer-initiated enhancements.

# Creating Enhancement Files

[This is preliminary documentation and subject to change.]

The following topics describe the steps necessary to create video enhancements:

- Creating Enhancement Pages
- Creating the Framework

After you have created the enhancement files, you can reference those files as you create the enhancement stream. An enhancement stream is a script that maps enhancement events, such as displaying a new page, to specific times in the enhanced show. For more information, see Creating an Enhancement Stream.

## Creating Enhancement Pages

[This is preliminary documentation and subject to change.]

Enhancement pages are HTML files that format the enhancement content for display. If you already know how to create interactive World Wide Web pages and you want to develop applications that receive interactive television content, you already have most of the information you need. Broadcast Architecture enhanced video is based on technologies you already know: Hypertext Markup Language (HTML), Microsoft® ActiveX™ controls, and Microsoft® Visual Basic® Scripting Edition (VBScript) or Microsoft® JScript™ development software.

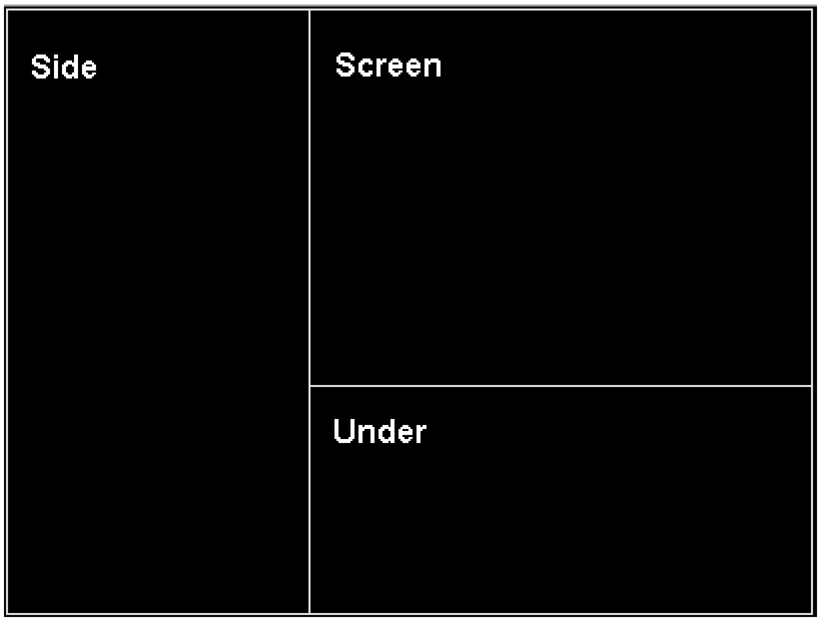
To locate additional information about designing and creating enhancements for shows, see Further Information on Data Services for the Client.

## Creating the Framework

[This is preliminary documentation and subject to change.]

Typically, enhancements are displayed in multiple frames. One of which plays the video, while the other frame(s) contain the enhancement content. Enhancement frames are built using exactly the same syntax that you use for building framed Web pages.

The example following creates a three-frame structure, as shown.



The HTML page following builds the three frames.

```
<HTML>
<head><Title>Simple Navigation</Title>
</head>
<BODY bgcolor="black" leftmargin=0 topmargin=0 link="#FFFFFF">
   <table width=800 height=600  cellpadding=0 cellspacing=0
   bordercolor="yellow" border=1>

      <tr><!-- Begin left frame (left 300 pixels) -->
         <td rowspan=2 width=300 height=600 >
         <IFRAME src="Side.htm" name="Side" width=300 height=600
            scrolling="no" frameborder=0 vspace=0 NORESIZE> </IFRAME>
         </td>

      <!-- Begin video screen frame (right 500 pixels) -->
         <td width=500 height=375 align="left" valign="top">
         <!-- The follow tag creates an instance of the Enhancement
              Video control -->
         <OBJECT
           ID=myEnhVideo
           CLASSID="clsid:a74e7f00-c3d2-11cf-8578-00805fe4809b"
           BORDER=0
           VSPACE=0
           HSPACE=0
           ALIGN=TOP
           HEIGHT=100%
           WIDTH=100%
         >
             <PARAM NAME="INTENT" VALUE=ENHANCE_VIDEO>
         </OBJECT>
      <tr><!-- Begin bottom-right frame -->
         <td>
         <IFRAME src="Under.htm" name="Under" width=500 height=225
          scrolling="no" frameborder=0 vspace=0 NORESIZE> </IFRAME>
         </td></tr>
</TABLE>
```

```
<!-- This is the Broadcast Architecture EnhCtrl Object -->
<OBJECT
CLASSID="clsid:3A263EF8-D768-11D0-911C-00A0C91F37E3"
WIDTH=0 HEIGHT=0 ID=EnhCtrl>
</OBJECT>

</BODY>
</HTML>
```

For more information about the object created in the preceding example, see Enhancement Video Control.

In addition to frames, you can also use overlays to create an enhancement layout. Overlays, also know as CSS positioning, are a feature of Dynamic Hypertext Markup Language (DHTML) that define the placement of elements on a page. You can use overlays in enhancement pages to position the video and enhancement content on the same page. Overlays provide better performance that HTML frames.

To locate more information about CSS positioning, see Further Information on Data Services for the Client.

## Creating an Enhancement Stream

[This is preliminary documentation and subject to change.]

An enhancement stream is a script that synchronizes enhancement events with specific times in the enhanced show. The script is written in stream compiler syntax and stored as a text file. If you are planning to broadcast an enhanced show, you will need to create an enhancement stream.

The enhancement stream coordinates enhancement actions with the video. For example, your script can ensure that files are transmitted prior to their display, triggers occur at specific times during the show, and that important files are repeatedly transmitted so they are always available to the viewer.

There are three methods that you can use to build an enhancement stream:

- Write the stream explicitly using stream compiler syntax and save it as a text file. This method requires fluency in the stream syntax language and advanced knowledge of the client system. You may still want to use a tool to validate your syntax and/or flatten the stream. For more information about the stream compiler syntax, see Flattening an Enhancement Stream or Further Information on Data Services for the Client.
- Use the enhancement stream editor to create an enhancement stream. This tool displays the enhancement stream as a time line and assists you in budgeting the available bandwidth. To locate more information on the enhancement stream editor, see Further Information on Data Services for the Client.
- Create a custom stream editor tailored to your needs, and use that to create an enhancement stream.

The order and composition of your enhancement stream depends on the content and structure of your enhanced show. However, you should keep in mind the following guidelines when composing an enhancement stream:

- Send an enhancement announcement for the enhanced show before it begins and repeatedly during the show. This loads the enhancement information into the Guide database, where it can be retrieved and used by TV Viewer.
- Schedule dependencies to be broadcast before the enhancement page that requires them.
- Ensure that files referred to by NavBase and NavFrame triggers are broadcast to the client before you broadcast the trigger. This ensures that you only redirect TV Viewer to display files that are available.
- Repeatedly transmit critical enhancement events. If the user tunes to the enhanced show in the middle of the episode, she or he may miss events broadcast earlier in the show. Repeatedly broadcasting enhancement files and triggers enables TV Viewer to receive and display enhancements even when the user was tuned to another channel for part of the episode.

## Flattening an Enhancement Stream

[This is preliminary documentation and subject to change.]

The Stream Compiler syntax can be written in either *high-level, unflattened* format or *low-level, flattened* format. Writing the stream in unflattened format makes the syntax compact and easy to understand. However before you can broadcast the stream using **ipsend**, you must flatten the stream to low-level format.

Using unflattened statements you can specify functionality such as 'transmit this trigger every 60 seconds' or 'transmit this file and all its dependencies' in a single statement. When the stream is flattened, each transmission required to implement those requests is specified explicitly in a separate statement.

For example, the following statement uses unflattened syntax to indicate that a file and its dependencies should be repeatedly broadcast at 60-second intervals throughout the show:

```
before 00:10:30.00 trigger ( 3 "Left" "Left_07.htm") repeat 60 until ShowLength;
```

When this statement is flattened, it is converted into the following set of statements (assuming that Pretty.gif is a dependency of Left_07.htm):

```
00:10:19:00 "Pretty.gif";
00:10:26:00 "Left_07.htm" only;
00:10:30:00 trigger 1 "Left_07.htm" only;
00:11:19:00 "Pretty.gif";
00:11:26:00 "Left_07.htm" only;
00:11:30:00 trigger 1 "Left_07.htm" only;
00:12:19:00 "Pretty.gif";
00:12:26:00 "Left_07.htm" only;
```

```
00:12:30:00 trigger 1 "Left_07.htm" only;
```

and so on.

You can use the **Events.Flatten** method of the Stream Compiler Objects to flatten an enhancement stream.

**Note** The stream compiler object library, Stream.dll, is not part of the software supporting the Broadcast Architecture Programmer's Reference. To locate this library, see Further Information on Data Services for the Client.

For example, the following code loads an enhancement stream from the file, EnhStr.txt, flattens it, and then saves the flattened stream to the file, EnhStrFlat.txt.

```
Dim evs As IEvents
Set evs = New Events
evs.Load("C:\EnhTools\EnhStr.txt")
evs.Flatten
evs.Store("C:\EnhTools\EnhStrFlat.txt")
```

## Broadcasting Enhancements

[This is preliminary documentation and subject to change.]

Once you have created your enhancement files and compiled and flattened an enhancement stream, the next step is to broadcast the enhancements and video to clients.

You can use the enhancement sender object, **ipsend**, to broadcast enhancement data, such as announcements, FTS data, and triggers, to clients. Enhancements can be broadcast over any transport that supports IP protocol, including the multicast routers and LAN networks.

The following function broadcasts events from an enhancement stream, using **ipsend** to transmit those events. This example assumes that a global instance of **ipsend**, isend, has already been created and connected to the outgoing IP streams.

```
Function SendEvent(e As IEvent) As String

    If e.IsTrigger Then

        'If the trigger is a NavFrame trigger and is missing the
        'initial '&' character, add it, otherwise do not
        If Mid$(e.Name, 1, 1) = "&" Or e.trigger <> 3 Then
           x$ = ""
        Else
           x$ = "&"
        End If

        'Transmit the trigger
        '(For triggers, Event.Name contains the trigger data)
        isend.SendTrigger e.Trigger, x$ + e.Name

    Else

        If e.IsAnnounce Then
            'Transmit the announcement
            '(For announcements, Event.Name contains the name and
            ' path of the announcement file)
            isend.SendAnnouncement e.Name
        Else
            'Transmit the file, using FTS
            '(For FTS events, Event.Name contains the source filename)
            'In this example the source and destination
            'filenames are the same.
            isend.SendFTSFile e.Name, e.Name
        End If

    End If

End Function
```

**Note** The enhancement stream must be flattened before the preceding function runs. For more information, see Flattening an Enhancement Stream. This step is not necessary if you have written the enhancement stream using only flat syntax.

To use a function such as **SendEvent** in an application, you could create a timer routine that compares the start time of the next event in an enhancement stream with the current time elapsed in the show. When the time for the event arrives, the timer routine could call **SendEvent** to broadcast the event.

## Receiving Enhancements on the Client

[This is preliminary documentation and subject to change.]

Computers that have the TV Viewer component of the Microsoft® Windows® 98 operating system installed can receive enhanced video. Enhancement announcements are automatically received and handled by the enhancement filter. If your enhancements need triggers or to receive FTS data, you can easily add this functionality by including the enhancement client controls in your enhancement pages.

The following lists the uses of the various client controls:

- To receive and handle triggers and FTS downloads, create an instance of the enhancement control, **EnhCtrl**, in one of your enhancement pages. For more information, see Receiving Triggers.
- To handle custom triggers on a page other than the one that contains **EnhCtrl**, create an instance of the user trigger object, **EnhUser**. For more information, see Handling User Triggers.

## Displaying Enhancement Pages in TV Viewer

[This is preliminary documentation and subject to change.]

TV Viewer is the television viewing component of Windows 98. Typically, this is the application in which the users will view your enhanced show.

When the user tunes TV Viewer to a new channel, it checks the Guide database to see whether the new show or channel is enhanced. If it finds enhancement information, and the user has not disabled enhancement viewing, TV Viewer uses the information to display the enhancement.

If you want your enhancements to be able to be displayed by TV Viewer, you must load information about the enhancement into the guide database. This is done with enhancement announcements.

An enhancement announcement is sent at some point before the client receives the enhancements, such as before the show is broadcast. It contains data about the enhancement, such as the enhancement identifier, the show reference of the show or channel that it enhances, and the filename of the initial enhancement page.

The Announcement Listener receives all the announcements sent to the client. It routes enhancement announcements to the enhancement filter. The filter in turn calls the **EnhLoaderProxy** object to load the enhancement data into the Guide database.

You can add an enhancement announcement to the enhancement stream in one of several ways:

- Use the Announcement Wizard feature of the enhancement stream editor. To locate more information on the enhancement stream editor, see Further Information on Data Services for the Client.
- Write the announcement specification in stream compiler syntax. For example, adding the following statement to the enhancement stream causes the announcement specified in Myshow.ann to be transmitted prior to the one-minute mark of a show.

  ```
  before 00:01:00:00 announcement "Myshow.ann";
  ```

- Use the stream compiler objects, or a tool based on those objects to schedule an announcement. For example, the following Visual Basic code creates an announcement identical to the one in the preceding example.

  ```
  'Add a new event using stream compiler syntax
  evs.AddText("before 00:01:00:00 ""Myshow.ann"" announcement;")
  ```

  In addition to adding a new event, you can use the stream compiler objects to change the text of an existing event to the announcement text.

```
'e1 is a previously-created enhancement event
evs.Add(60, "First Announcement")
Set e1 = evs.LastAdd
...
'Set the text of the e1 to specify the announcement
e1.Text = "before 00:01:00:00 ""Myshow.ann"" announcement;"
```

**Note** The format of the announcement file, Myshow.ann in the preceding examples, is described in the topic, Enhancement Announcement Format.

For more information about the stream compiler syntax, see Stream Compiler Objects and Further Information on Data Services for the Client.

## Handling Triggers

[This is preliminary documentation and subject to change.]

To receive broadcast video, broadcast data, announcements, and triggers, the HTML pages on the broadcast client must contain special ActiveX controls that tune to the appropriate network channels and respond appropriately. These controls are included with Broadcast Architecture

There are two controls that you can use to generate trigger events: the enhancement control and the user trigger control, **EnhUser**. **EnhCtrl** is the enhancement control. It handles Broadcast Architecture triggers and sends an event for each user trigger it receives. **EnhUser** does not handle any triggers, it only sends an event when the client receives a user trigger. **EnhUser** is useful in situations where you need to generate user-trigger events on a page in the enhancement other than the one that contains **EnhCtrl**.

For details on how to use these controls in a Web page and implement scripts that handle the trigger events, see the following topics:

- Receiving Triggers
- Handling Broadcast Architecture Triggers
- Handling User Triggers

### Receiving Triggers

[This is preliminary documentation and subject to change.]

In order for your enhancement pages to receive triggers, you must create an instance of the enhancement control, **EnhCtrl**. The enhancement control, **EnhCtrl**, monitors the enhancement IP stream for triggers. This control interacts with TV Viewer or the Web browser that is hosting it to handle Broadcast Architecture triggers.

To create an instance of **EnhCtrl** in an enhancement or Web page, use the following HTML:

```
<!-- Creating the EnhCtrl Object -->
<OBJECT
CLASSID="clsid:3A263EF8-D768-11D0-911C-00A0C91F37E3"
WIDTH=0 HEIGHT=0 ID=myEnhCtrl>
</OBJECT>
```

**Note** There must be only one of **EnhCtrl** on any set of framed enhancement pages. Multiple instances of this control causes multiple navigation calls and other errors. For example, if there are two instances of the enhancement control, a single NavBase trigger will cause the specified page to load twice.

## Handling Broadcast Architecture Triggers

[This is preliminary documentation and subject to change.]

Broadcast Architecture defines a set of triggers with specific functionality. These triggers command the client to do such actions as display new content in a frame, display the enhancement's starting page, or receive an FTS download.

The enhancement control, **EnhCtrl**, automatically handles Broadcast Architecture triggers, performing the specified action. For example, if the client receives a trigger to display change the content displayed in a specific frame, **EnhCtrl** interacts with TV Viewer or the Web browser hosting it, to display the new content.

For more information about the types of triggers defined by Broadcast Architecture, see Enhancement Triggers.

## Handling User Triggers

[This is preliminary documentation and subject to change.]

User triggers open essentially unlimited possibilities for control over your enhanced production. Because user triggers can invoke subroutines that you create in VBScript or JScript, you can perform any type of action, including the use of client-side decision branches and dynamic generation of HTML, based on receipt of a particular trigger message.

You can use either **EnhCtrl** or **EnhUser** in a page to receive user triggers and send an event that calls a VBScript subroutine. If an instance of **EnhCtrl** is already present in the page, you do not need to use **EnhUser**. However, you can create other pages within the enhanced show that include instances of the **EnhUser** control. (Only one instance of **EnhCtrl** can run at once, but any number of instances of **EnhUser** can run simultaneously.)

The following code example defines a subroutine that is invoked when the control receives a user trigger. The example works on the assumption that the following code appears in the same file **EnhCtrl** is embedded in.

The subroutine is initiated by the occurrence of a user trigger. The subroutine first examines the integer key and performs a case statement to determine the appropriate action based on the key. In this example, if the key is 1001 the trigger is assumed to refer to a Macromedia Flash animation elsewhere in the frame structure. In this case, the script then examines the string to determine what action to take. The script sets the animation to the frame number specified by the contents of the string.

```
<SCRIPT LANGUAGE="VBScript">
<!--
Sub myEnhCtrl_onTrigger(lkey, lpszTrigger)
    Select Case lKey
        Case 1001  ' Set skyline animation
            if lpszTrigger = "Day" then
                parent.frames(0).MainAni.framenum=10
            elseif lpszTrigger = "Dusk" then
                parent.frames(0).MainAni.framenum=11
            elseif lpszTrigger = "Night" then
                parent.frames(0).MainAni.framenum=12
            end if
        Case 1002
            MsgBox "Another case"
        Case Else
        End Select
End Sub

-->
</SCRIPT>
```

Other user trigger numbers (1,002 and so on) might be used for other effects. User triggers can be numbered from 1,000 to 2,147,483,647.

## Controlling Navigation in Enhancements

[This is preliminary documentation and subject to change.]

*Navigation* refers to directing HTML pages to appear in various frames, either under control of the broadcast show or of the viewer. Loosely, navigation refers to creating a set of frames and controls that interact with one another to create an enhanced show.

The following topics provide an overview of the options you have for navigation within enhanced video:

## Synchronous Navigation Using Triggers

[This is preliminary documentation and subject to change.]

To illustrate synchronous navigation using triggers, suppose you want a series of different enhancements to appear in the Side frame, shown in Creating the Framework, at specific moments during the show. To produce this effect, you create a page that responds to a *trigger*. The trigger is broadcast at the proper moment during the show.

The type of trigger that produces the necessary action is called a *NavFrame* trigger. A NavFrame trigger specifies both a Uniform Resource Locator (URL) to display and the frame in which to display it. To respond to the NavFrame trigger, you embed the object **EnhCtrl** into the Web page, as shown in Creating the Framework. The ActiveX control **EnhCtrl** automatically performs the navigation when it receives a trigger of this type. For more information, see Enhancement Client Controls.

Once **EnhCtrl** is embedded, the process is complete. Assuming that the NavFrame triggers are broadcast with the target frame specified as Side, the specified pages appear in those frames during the show.

## Handling Viewer-Initiated Navigation

[This is preliminary documentation and subject to change.]

Some shows support two different types of enhancements. *Synchronous* enhancements appear when triggers are received during the course of the show. *Viewer-initiated* enhancements are caused by the person watching the show. For example, when the viewer clicks **Cast** in the menu control, an animation featuring the cast of the show appears in the left frame, replacing the current synchronous enhancement.

Suppose a new synchronous enhancement is triggered to appear in the left frame while the viewer interacts with user-initiated enhancements in the same frame. The **EnhCtrl** control handles this situation by saving the Uniform Resource Locator (URL) of the synchronous enhancement but not displaying the enhancement immediately. When the viewer switches back to watching the show, the appropriate synchronous enhancement appears.

To control whether synchronous enhancements should appear, use the **EnhCtrl.FrameSync** property. The syntax is:

```
EnhCtrl.FrameSync(framename) = [True|False]
```

For example, the following statement sets the control so that if any subsequent NavFrame trigger occurs with the Side frame as the destination, the file name of the synchronous trigger is retained but not displayed:

```
EnhCtrl.FrameSync("Side") = False
```

The next statement sets the control so that the file most recently named by a NavFrame trigger is immediately displayed, and so that subsequent NavFrame triggers that name the Side frame as the destination immediately display the file:

```
EnhCtrl.FrameSync("Side") = True
```

The following HTML example shows how synchronous enhancements can be turned on and off under user control. Because the buttons exist in Under.htm, which is a subframe of the page that contains the control, the script must reference the control in the outer page by preceding the control name with `parent`.

```
<form method="POST" NAME="TestForm">
    <input type=button maxlength=256 name="Synchronous" value="Synchronous">
        <SCRIPT FOR="Synchronous" EVENT="onClick" LANGUAGE="VBScript">
            ' This button turns synchronous enhancements on.
            parent.EnhCtrl.FrameSync("Side") = True
        </SCRIPT>
    <input type=button maxlength=256 name="Viewer" value="Viewer">
        <SCRIPT FOR="Viewer" EVENT="onClick" LANGUAGE="VBScript">
            ' This button turns synchronous enhancements off and
            ' displays a viewer navigation page.
            parent.EnhCtrl.FrameSync("Side") = False
            parent.frames(0).location.href="Viewer.htm"
        </SCRIPT>
```

# Enhancement Reference

[This is preliminary documentation and subject to change.]

The following topics provide reference information about the enhancement-related objects and libraries:

- Enhancement Announcement Format describes the format used in enhancement announcements.
- Enhancement Triggers describes the types and formats of enhancement triggers.
- Enhancement Daemon 1.0 Type Library discusses the enhancement control and other objects that can be used in enhancement Web pages.
- **EnhLoaderProxy** describes the object called by the enhancement filter to load enhancement information into the Guide database.

- Stream Compiler Object Library describes the programmable COM objects exposed by the compiler that creates and edits the enhancement stream.
- **ipsend** details an object you can use to transmit enhancement events.

**Note** The preceding object references are documented using syntax for the Microsoft® Visual Basic® development system . However, because the enhancement objects implement COM, you can use them from any COM-supporting language, including Java and the Microsoft® Visual C++® development system.

## Enhancement Announcement Format

[This is preliminary documentation and subject to change.]

Enhancement announcements are formatted using Session Description Protocol (SDP). For more information, see Announcement Format. This topic discusses only the SDP fields used by enhancement announcements.

The following is a sample enhancement announcement for a future show:

```
v=0
o=enhfilt 248 56132 IN IP4 157.55.106.31
s=Fresh Prince Enhanced
c=IN IP4 231.31.17.1
t=31557167 31449234
a=EnhID:{9E2E8B20-083E-11d1-898F-00C04FBBDEBC}
a=ShowRef:1997/7/29!26673/7175/9690!0:30!0!0!0!0!9690!9690!4096!7040!''!'CBUT'!3!54
a=PreloadURL:show\default.htm
a=FutureEnh:
m=data 3456 udp 0
```

where

**v =**
Announcement protocol version.

**o =**
Owner of the announcement, in this case the enhancement filter, and session identifier.

**s =**
Name of the enhancement. This string should be used as the enhancement title in any user interface relating to the enhancement.

**c =**
Connection information.

**t =**
Time the session is active.

**a =**
Attribute fields, of which there can be zero or more. The enhancement announcement format specifies the following attribute fields:
**a = EnhID**

Identifies the announcement as an enhancement announcement and specifies a unique identifier for the enhancement.

**a = ShowRef**

Specifies the show reference of the enhanced episode or episodes. This field is required for future announcements. If this field is not present, the filter assumes the announcement is for a current broadcast. For more information, see Show Reference Format.

**a = PreLoad**

Specifies the Uniform Resource Locator (URL) of the HTML file that contains the layout of the enhancement. This field is optional for a future announcement and is not used for current announcements. If you do not specify a complete path, such as C:\MyEnhance\, the URL is resolved relative to C:\Program Files\TV Viewer\Layouts. For example, if you specify Cspan\Cspan.htm, this path is resolved to C:\Program Files\TV Viewer\Layouts\Cspan\Cspan.htm.

**a=FutureEnh**

Indicates that the announcement is for a future enhancement. This field is required for future enhancements and is not used for current enhancements.

**m =**

Specifies the media name and transport address.

## Enhancement Triggers

[This is preliminary documentation and subject to change.]

Each type of trigger sent to the enhancement control has a specific format associated with it. The basic trigger format is a string that contains two parts, a key or numerical identifier, and the trigger data. The key is separated from the data by white space, typically a space or tab character.

"*Key TriggerData*"

For example, the following string might be specified in a NavBase trigger:

```
"2 http://www.microsoft.com/default.htm"
```

The following table lists the trigger formats currently supported.

| Key | Name | Description |
|---|---|---|
| 0 | Error | No action is taken. |
| 1 | FTSData | Receives File Transfer Service (FTS) data at the specified address and port. |
| 2 | NavBase | Displays the specified page as the top-level frame. |

| | | |
|---|---|---|
| 3 | NavFrame | Displays the specified page in the specified frame. |
| 4 – 999 | Reserved | Key values reserved for future trigger functionality. |
| 1,000 and up | UserTrigger | Sends user trigger event. |

## FTSData Trigger

[This is preliminary documentation and subject to change.]

FTSData triggers indicate that File Transfer Service (FTS) data, such as an HTML enhancement file, is about to be transmitted. They are formatted as shown following

**"1** *strIPAddress***"**

where *strIPAddress* is a **String** that contains the IP address and port that the FTS data will be transmitted on. This address and port must be in the following format: *xxx***.***xxx***.***xxx***.***xxx***:***yyyy*, where *xxx***.***xxx***.***xxx***.***xxx* is the IP address, and *yyyy* specifies the port.

When the enhancement control receives a FTSData trigger, it creates an instance of an internal object to receive the file. Multiple files may be received at the same time on different IP address and port combinations.

All files transmitted on the specified IP address and port are placed into the spool directory.

## NavBase Trigger

[This is preliminary documentation and subject to change.]

NavBase triggers store preload URL data in the enhancement control for later use. NavBase triggers are formatted as shown following

**"2** *strPreloadURL***"**

where *strPreloadURL* is a **String** that contains the location of the enhancement's initial HTML page. This location can be either a fully specified URL, or a URL relative to the spool directory, C:\Program Files\TV Viewer\Enhspool\.

The NavBase trigger, when received by the **EnhCtrl** control, navigates to the highest-level window of the Web browser to the specified URL. (This functionality produces the same result as setting

**TARGET="_top"** in an HREF link.) This trigger can be used to reset the entire enhanced show.

However, usually you do not need this trigger. This trigger usually is not necessary because the Program Guide contains the base URL for each show and updates the container automatically when the viewer changes channels, or when a new show begins on the same channel.

## NavFrame Trigger

[This is preliminary documentation and subject to change.]

NavFrame triggers cause the enhancement control to display a new HTML page in the specified frame. They are formatted as shown following

**"3 &***strFrame***&***strURL***"**

where *strFrame* is a **String** that contains the name of the frame to display the new content, and *strURL* is a **String** that contains the location of the HTML page to display. This location can be either a fully specified URL, or a URL relative to the spool directory, C:\Program Files\TV Viewer\Enhspool\.

Using a NavFrame trigger, you can sequence complex enhanced Web content without any scripting.

The results of this trigger depend on the state of the FrameSync property of the **EnhCtrl** control. When this property is set to False, the control does not navigate immediately to the specified URL but does save the URL. When this property is once again set to True, the control navigates to the URL most recently specified.

The directory for URLs specified by Navigate Base and NavFrame triggers is the spool directory.

## UserTrigger Trigger

[This is preliminary documentation and subject to change.]

User triggers are custom triggers sent by the broadcast provider. They are formatted as show following

**"***lKeyID strData***"**

where *lKeyID* is a **Long** that indicates the trigger identifier, whose value is defined by the content provider and must be greater than 1,000, and *strData* is a **String** that contains the trigger information. The trigger information data is parsed and used by the event handler that processes the user trigger.

# Enhancement Daemon 1.0 Type Library

[This is preliminary documentation and subject to change.]

The Enhancement daemon 1.0 type library, Entrig.dll, provides the following objects:

- **EnhCtrl**, the enhancement control.
- **EnhUser**, the user trigger object.

## EnhCtrl

[This is preliminary documentation and subject to change.]

**EnhCtrl**, provided by Entrig.dll, is the enhancement control. It receives an incoming Internet Protocol (IP) data stream, responds to navigation triggers, and makes user triggers available to scripted routines. This control must appear in one of the files belonging to the enhanced show.

The enhancement control provides the following properties.

| Property | Description |
|---|---|
| **Address** | IP address used to connect to the enhancement stream. |
| **EnableErrorMsg** | Value that specifies whether the enhancement control displays error messages to the user. |
| **EnableNavBase** | Value that specifies whether the enhancement control responds to NavBase triggers |
| **EnhancedShow** | Value that specifies whether the enhancement control has the information that enhancements are currently being displayed. |
| **FrameSync** | Value set to enable or disable synchronous triggers. |
| **NetCard** | Network card address used to connect to the enhancement stream. |
| **Port** | Port used to connect to the enhancement stream. |
| **TriggerSource** | Reference to the trigger source object. Set **EnhUser.TriggerSource** equal to this value to enable the user trigger object to receive triggers. |

The enhancement control provides the following methods.

| Method | Description |
|---|---|
| **InitiateConnection** | Connects the enhancement control to the trigger stream. |
| **SendTrigger** | Generates a "virtual" trigger. This method is typically used for testing. |

The enhancement control provides the following events.

| Event | Description |
|---|---|
| BeforeNavFrame | The control is about to navigate a frame to a new Web page in response to a NavFrame trigger. |
| DisplayOverlay | This event is always sent following an OnNewOverlay event. |
| HideOverlay | This event is always sent preceding an OnRemoveOverlay event. |
| OnNewOverlay | The control has received a **NavBase** trigger that specifies an overlay. |
| OnRemoveOverlay | The control has received a trigger navigating it back to full-screen video or the program guide. |
| OnTrigger | The control has received a user trigger from the Announcement Listener. |

**Remarks**

There must be only one instance of this control on any set of enhancement pages. Multiple instances of this control causes multiple navigation calls. For example, if there are two instances of the enhancement control, a single NavBase trigger will cause the specified page to load twice.

The enhancement control operates in any of three modes, normal, loopback, and authoring. In normal mode, the client monitors for triggers and enhancement files on an IP multicast address. Error conditions such as missing files are ignored. In loopback mode, the client monitors for triggers and enhancement files on a loopback IP address. In other words, the source is the broadcast client itself. Error conditions such as missing files are ignored. In authoring mode, the client monitors the loopback address, and error conditions such as missing files are reported with error messages.

The mode is determined by the situation in which the enhancement client control is created. If the control is created in TV Viewer and IP information is provided, the control runs in normal mode. If the control is created in TV Viewer but no IP information is given, the control runs in loopback mode. Finally, if the control is created under a stand-alone instance of Internet Explorer, the control runs in authoring mode.

**See Also**

**EnhUser**

### Examples

The following HTML creates an instance of a **EnhCtrl** object on a Web or enhancement page.

```
<!-- This is the Broadcast Architecture EnhCtrl Object -->
    <OBJECT
    CLASSID="clsid:3A263EF8-D768-11D0-911C-00A0C91F37E3"
    WIDTH=0 HEIGHT=0 ID=myEnhCtrl>
    </OBJECT>
```

## EnhCtrl.Address

[This is preliminary documentation and subject to change.]

The **Address** property specifies the multicast IP address used to connect to the enhancement stream.

### Syntax

*object*.**Address** [ = *strAddress*]

### Parts

*object*
  Object expression that resolves to the **EnhCtrl** object.
*strAddress*
  **String** that specifies the IP address on which the enhancement is received. This address should be in the format *xxx*.*xxx*.*xxx*.*xxx*, for example 255.255.255.255.

### Remarks

This property should only set if your application does not use TV Viewer. If you call the **ITVViewer::Tune** method, the address parameter specified during the method call overrides the value set in **EnhCtrl.Address**.

## EnhCtrl.EnableErrorMsg

[This is preliminary documentation and subject to change.]

The **EnableErrorMsg** property indicates whether the enhancement control displays error messages to the user.

### Syntax

```
object.EnableErrorMsg [ = lEnable ]
```

### Parts

*object*
  Object expression that resolves to the **EnhCtrl** object.

*lEnable*
  **Long** that indicates whether error messages should be enabled. If this value is non-zero, error messages are enabled. If this value is zero, they are not.

## EnhCtrl.EnableNavBase

[This is preliminary documentation and subject to change.]

The **EnableNavBase** property indicates whether to respond to NavBase triggers.

### Syntax

```
object.EnableNavBase [ = lNavBase ]
```

### Parts

*object*
  Object expression that resolves to the **EnhCtrl** object.

*lEnable*
  **Long** that indicates whether NavBase triggers should be enabled. If this value is non-zero, error messages are enabled. If this value is zero, they are not.

## EnhCtrl.BeforeNavFrame

[This is preliminary documentation and subject to change.]

The BeforeNavFrame event occurs before the enhancement control navigates a frame to a new HTML page, typically in response to a NavFrame trigger.

### Syntax

```
Private Sub object_BeforeNavFrame( bstrFrame, bstrURL, lCancel )
```

### Parameters

*object*
: Object expression that resolves to the **EnhCtrl** object.

*pbstrFrame*
: **String** that contains the name of the HTML frame in which the Web page should appear.

*pbstrURL*
: **String** that contains the URL of the Web page to display. Typically, this URL points to a page stored in the enhancements directory of the user's computer.

*bCancel*
: **Long** that receives a value from the event handler that indicates whether to cancel the frame navigation. If this value is non-zero, the navigation should be canceled. If it is zero, it should not be canceled.

### Remarks

Your application can handle this event to prevent the enhancement control from navigating to the HTML page specified in the NavFrame trigger.

## EnhCtrl.EnhancedShow

[This is preliminary documentation and subject to change.]

The **EnhancedShow** property specifies whether enhancements are currently being displayed.

### Syntax

```
object.EnhancedShow [ = lEnhanced ]
```

### Parts

*object*
: Object expression that resolves to the **EnhCtrl** object.

*lEnhanced*
: **Long** that indicates whether the enhancement control should display enhancements. If this value is non-zero, the control displays enhancements. If it is zero, it hides enhancements.

### Remarks

The default value for this property is **True**. If you set **EnhancedShow** to **False**, the enhancement control attempts to initialize TV Viewer with the information required to display enhancements for the current show.

Because setting **EnhancedShow** to **False** updates the TV Viewer database, potentially disrupting any currently displayed enhancements, it is recommended that you not set this property.

## EnhCtrl.FrameSync

[This is preliminary documentation and subject to change.]

The **FrameSync** property indicates whether the enhancement control should display synchronous, or content provider–triggered, enhancements during the course of the show.

### Syntax

```
object.FrameSync(bstrFrame) [ = lMode ]
```

### Parts

*object*
: Object expression that resolves to the **EnhCtrl** object.

*bstrFrame*
: **String** that specifies the HTML frame to which the **FrameSync** setting applies.

*lMode*
: **Long** that indicates whether the enhancement control should display synchronous enhancements. If this value is non-zero, the control displays synchronous enhancements. If it is zero, the control hides them.

### Remarks

If a user clicks on a link in an enhancement page and synchronous triggers are not disabled, a synchronous trigger can fire, causing a page other than the user-selected page to display. By turning synchronous triggers off during user interactions, you can avoid this type of navigational conflict.

## EnhCtrl.InitiateConnection

[This is preliminary documentation and subject to change.]

The **InitiateConnection** method connects the enhancement control to the trigger stream.

### Syntax

```
object.InitiateConnection()
```

### Parameters

*object*
Object expression that resolves to the **EnhCtrl** object.

### Remarks

The connection is made using the IP address, network card, and port specified during the call to the **ITVViewer::Tune** method. If your application does not use TV Viewer, you can set these values in the **EnhCtrl.Address**, **EnhCtrl.Port**, and **EnhCtrl.NetCard** properties.

## EnhCtrl.NetCard

[This is preliminary documentation and subject to change.]

The **NetCard** property specifies the network card address used to connect to the enhancement stream.

### Syntax

```
object.NetCard [ =  strNetCard]
```

### Parts

*object*
Object expression that resolves to the **EnhCtrl** object.
*strNetCard*
**String** that contains the network card address. This address should be in the format *xxx***.***xxx***.***xxx***.***xxx*, for example 125.125.125.125.

### Remarks

This property should only set if your application does not use TV Viewer. If you call the **ITVViewer::Tune** method, the network card parameter specified during the method call overrides the value set in **EnhCtrl.NetCard**.

# EnhCtrl.Port

[This is preliminary documentation and subject to change.]

The **Port** property specifies the port used to connect to the enhancement stream.

## Syntax

```
object.Port [ = iPort]
```

## Parts

*object*
Object expression that resolves to an **EnhCtrl** object.

*iPort*
**Integer** that specifies the port, for example 80.

## Remarks

This property should only set if your application does not use TV Viewer. If you call the **ITVViewer::Tune** method, the port parameter specified during the method call overrides the value set in **EnhCtrl.Port**.

# EnhCtrl.TriggerSource

[This is preliminary documentation and subject to change.]

The **TriggerSource** property contains a reference to the trigger source object. This is an internal object that parses triggers received by the client.

## Syntax

```
object.TriggerSource [ = unkSource]
```

## Parts

*object*
Object expression that resolves to an **EnhCtrl** object.

*unkSource*

  **Unknown** that contains a reference to the internal trigger source object.

### Remarks

You can use this property to connect a user trigger object to the trigger stream. Simply set **EnhUser.TriggerSource** = **EnhCtrl.TriggerSource**.

### See Also

**EnhUser.TriggerSource**

## EnhCtrl.SendTrigger

[This is preliminary documentation and subject to change.]

The **SendTrigger** method causes the enhancement control to act as if it has received a trigger of the specified type.

### Syntax

```
object.SendTrigger(strTrigger)
```

### Parameters

*object*

  Object expression that resolves to the **EnhCtrl** object.

*strTrigger*

  **String** that contains the trigger. This trigger should be in standard trigger format. For more information, see Enhancement Triggers.

### Remarks

This method can be used to test enhancement client applications when usual trigger broadcasting is not available.

## EnhCtrl.DisplayOverlay

[This is preliminary documentation and subject to change.]

The DisplayOverlay event always occurs following an OnNewOverlay event.

**Syntax**

```
Private Sub object_DisplayOverlay()
```

**Parts**

*object*
Object expression that resolves to the **EnhCtrl** object.

**See Also**

**EnhCtrl.**HideOverlay, **EnhCtrl.**OnNewOverlay

# EnhCtrl.HideOverlay

[This is preliminary documentation and subject to change.]

The HideOverlay event always occurs preceding an OnRemoveOverlay event.

**Syntax**

```
Private Sub object_HideOverlay()
```

**Parts**

*object*
Object expression that resolves to the **EnhCtrl** object.

**See Also**

**EnhCtrl.**DisplayOverlay, **EnhCtrl.**OnRemoveOverlay

# EnhCtrl.OnNewOverlay

[This is preliminary documentation and subject to change.]

The OnNewOverlay event occurs when the enhancement control receives a **NavBase** trigger that specifies an overlay page.

### Syntax

```
Private Sub object_OnNewOverlay(bstrOverlayURL As String, _
bstrOverlayCSS As String)
```

### Parts

*object*
Object expression that resolves to the **EnhCtrl** object.

*bstrOverlayURL*
String that specifies the URL of the DHTML overlay page.

*bstrOverlayCSS*
String that contains the style sheet of the overlay page.

## EnhCtrl.OnRemoveOverlay

[This is preliminary documentation and subject to change.]

The OnRemoveOverlay event occurs when the **EnhCtrl** object receives a trigger that causes it to navigate from an overlay back to full-screen video or the program guide.

### Syntax

```
Private Sub object_OnRemoveOverlay()
```

### Parts

*object*
Object expression that resolves to the **EnhCtrl** object.

## EnhCtrl.OnTrigger

[This is preliminary documentation and subject to change.]

The OnTrigger event occurs when the **EnhCtrl** object receives a user-defined trigger.

### Syntax

```
Private Sub object_OnTrigger(lKey As Long, pbstrData As String)
```

### Parts

*object*
    Object expression that resolves to the **EnhCtrl** object.
*lKey*
    **Long** that indicates the key, or identifier, of the user trigger. Valid values for this parameter are defined by the broadcast content provider that created the trigger.
*pbstrData*
    **String** that contains the user trigger data.

### Remarks

A user trigger is defined by the content provider and is transmitted to the client. By implementing event handlers in the enhancement Web pages, the broadcast client can support these provider-specific user triggers.

User triggers can also be intercepted and sent by the **EnhUser** object.

## EnhUser

[This is preliminary documentation and subject to change.]

**EnhUser**, provided by Entrig.dll, is the user trigger object. This object sends an event when it receives a user trigger. User triggers are defined by the content provider and can be used to implement triggers not provided by Broadcast Architecture. A user-defined trigger includes two parameters, an integer key and a string. The meaning of the integer and the string parameter depends on the broadcast content provider's implementation of the trigger.

You can use either an **EnhCtrl** or **EnhUser** control in a page to receive user triggers and send user trigger events. One advantage of using **EnhUser** is that multiple instances of **EnhUser** can run in different files within the enhanced show, whereas only one instance of **EnhCtrl** can exist on the client.

For example, one instance of **EnhUser** may be running in the left frame and respond only to triggers that apply to that frame. At the same time, another instance of the control in the bottom frame might respond to triggers that apply to that frame.

**EnhUser** provides a single event, OnTrigger, which the control sends when it receives a user trigger. EnhUser also has one property, **TriggerSource**, which contains a reference to the internal trigger source object.

### Examples

The following HTML creates an instance of a **EnhUser** object on a Web or enhancement page.

```
<!-- Creating the EnhUser Object -->
<OBJECT
CLASSID="clsid:3A263EFA-D768-11D0-911C-00A0C91F37E3"
WIDTH=0 HEIGHT=0 ID=myEnhUser>
</OBJECT>
```

## EnhUser.OnTrigger

[This is preliminary documentation and subject to change.]

The OnTrigger event occurs when the **EnhUser** object receives a user trigger.

### Syntax

```
Private Sub object_OnTrigger(lKey As Long, pbstrData As String)
```

### Parameters

*object*
  Object expression that resolves to the **EnhUser** object.

*lKey*
  **Long** that indicates the key, or identifier, of the user trigger. This key is defined by the content provider that sent the trigger. Valid key identifiers are those in the range 1,000 and above. Values less than 1,000 are reserved for internal use by Broadcast Architecture.

*pbstrData*
  **String** that contains the user trigger data.

### Remarks

A user trigger is defined by the content provider and is transmitted to the client. By implementing event handlers in the enhancement Web pages, the broadcast client can support these provider-specific user triggers.

User triggers can also be intercepted and sent by the **EnhCtrl** object.

## EnhUser.TriggerSource

[This is preliminary documentation and subject to change.]

The **TriggerSource** property contains a reference to the trigger source object. This is an internal object that parses triggers received by the client.

**Syntax**

```
object.TriggerSource [ = unkSource]
```

**Parts**

*object*
  Object expression that resolves to an **EnhUser** object.
*unkSource*
  **Unknown** that contains a reference to the internal trigger source object.

**Remarks**

You can use this property to connect a user trigger object to the trigger stream. Simply set **EnhUser.TriggerSource** = **EnhCtrl.TriggerSource**.

**See Also**

**EnhCtrl.TriggerSource**

## EnhLoaderProxy

[This is preliminary documentation and subject to change.]

The **EnhLoaderProxy** object, implemented in EnhLoad.exe, loads and deletes enhancement information in the Guide database. The enhancement filter calls this object to load the data it receives from announcements.

**EnhLoaderProxy** provides the following methods.

| Method | Description |
|---|---|
| **LoadEnhancement** | Loads information about an enhancement into the Guide database. |
| **DeleteEnhancement** | Deletes information about an enhancement from the Guide database. |

**EnhLoaderProxy** provides the following events.

| Event | Description |
|---|---|
| GetPreloadURL | **EnhLoaderProxy** has received a current enhancement announcement that does not contain a preload URL. |
| GetShowRef | **EnhLoaderProxy** has failed to load a current enhancement announcement based on identification information contained in the announcement. |
| SetMulticastIP | **EnhLoaderProxy** has matched an announcement and set the multicast IP address to the value specified in the announcement. |

**Remarks**

**EnhLoadProxy** uses the enhancement-loading methods of Television System Services (TSS) to load the enhancement data.

**See Also**

Enhancement Filter

## EnhLoaderProxy.DeleteEnhancement

[This is preliminary documentation and subject to change.]

The **DeleteEnhancement** method deletes the specified enhancement information from the Guide database.

**Syntax**

```
object.DeleteEnhancement(strID)
```

### Parameters

*object*
: Object expression that resolves to the **EnhLoaderProxy** object.

*strID*
: Unique identifier of the enhancement.

### Remarks

This method is basically a wrapper for **ITelevisionServices::DeleteEnhancementFromID**.

## EnhLoaderProxy.GetPreloadURL

[This is preliminary documentation and subject to change.]

The GetPreloadURL event occurs when the **EnhLoaderProxy** object receives a current enhancement announcement that does not contain a preload URL.

### Syntax

```
Private Sub object_GetPreloadURL(dateReceived As Date, _
 pbstrPreloadURL As String, pblsOverlay as Long, pbLoud As Long)
```

### Parts

*object*
: Object expression that resolves to the **EnhLoaderProxy** object.

*dateReceived*
: **Date** that specifies when the enhancement control received the NavBase trigger.

*pbstrPreloadURL*
: **String** that contains the location of the enhancement's initial HTML page. This location can be either a fully specified URL, or a URL relative to the spool directory, C:\Program Files\TV Viewer\Enhspool\.

*pblsOverlay*
: **Long** that indicates whether the preload URL is an overlay. If this value is non-zero, the URL specified by *pbstrPreloadURL* is an overlay. If it is zero, it is not.

*pbLoud*
: **Long** that indicates whether to notify TV Viewer of the change in the viewing state. If this value is non-zero, TV Viewer is notified. If it is zero, TV Viewer is not notified.

### Remarks

If the **EnhLoaderProxy** is called to load data from an announcement that does not contain a preload URL, it requests the preload URL information for the currently-displayed show from the enhancement control. The content provider can set the preload URL value that the enhancement control returns by sending a NavBase trigger.

## EnhLoaderProxy.GetShowRef

[This is preliminary documentation and subject to change.]

The GetShowRef event occurs when **EnhLoaderProxy** fails to load a current enhancement announcement based on identification information contained within the announcement, such as the show identifier or show reference.

### Syntax

```
Private Sub object_GetShowRef(dateReceived As Date, _
 lTuningSpace As Long, sChannel As Integer)
```

### Parameters

*object*
  Object expression that resolves to the **EnhLoaderProxy** object.
*dateReceived*
  **Date** that specifies when the enhancement control received the show reference trigger.
*lTuningSpace*
  **Long** that contains the tuning space of the current show.
*sChannel*
  **Integer** that contains the channel number of the current show.

### Remarks

**EnhLoaderProxy** requests this information after it fails to load data into the Guide database for a current enhancement announcement. **EnhLoaderProxy** creates the show reference by using the time the announcement was received and the tuning space and channel to which TV Viewer is tuned, as specified by the event.

## EnhLoaderProxy.LoadEnhancement

[This is preliminary documentation and subject to change.]

The **LoadEnhancement** method loads enhancement information into the Guide database.

### Syntax

```
object.LoadEnhancement(strID, strShowID, strShowRef, strTitle, _
 strNetCard, strMultiIP, iPort, strPreloadURL, dateExpire, _
 dateReceived, lShowLength, bCurrentShow)
```

### Parameters

*object*
Object expression that resolves to the **EnhLoaderProxy** object.

*strID*
Unique identifier for the enhancement. This identifier is specified in the owner (**o=**) field of the SDP announcement.

*strShowID*
Reserved for future use.

*strShowRef*
**String** that contains the show reference. If the announcement is for a show being currently broadcast, this value may be NULL. If the *strShowRef* parameter is NULL, **EnhLoaderProxy** gets the show reference of the current show from **EnhCtrl**.

*strTitle*
**String** that contains the enhancement title. This value is displayed by TV Viewer when an episode has more than one enhancement available. When the user clicks the enhancement icon in the channel banner, TV Viewer displays a list of the enhancement titles, in which the user can click which enhancement to display, for example **Fresh Prince Enhanced**

*strNetCard*
**String** that contains the network card address used to connect to the enhancement stream. This address should be in the format *xxx***.***xxx***.***xxx***.***xxx*, for example 125.125.125.125.

*strMultiIP*
**String** that contains the IP multicast address used to connect to the enhancement stream. This address should be in the format *xxx***.***xxx***.***xxx***.***xxx*, for example 255.255.255.255.

*iPort*
**Integer** that specifies the port used to connect to the enhancement stream, for example 10024.

*strPreloadURL*
URL of the HTML file that contains the enhancement layout. If you do not specify a complete path, such as C:\MyEnhance\, the URL is resolved relative to C:\Program Files\TV Viewer\Layouts. For example, if you specify *strPreloadURL* as Cspan\Cspan.htm, it resolves to C:\Program Files\TV Viewer\Layouts\Cspan\Cspan.htm.

If the announcement is for a show being currently broadcast, this value may be NULL. If *strPreloadURL* is NULL, **EnhLoaderProxy** gets the preload URL of the current show from **EnhCtrl**.

*dateExpire*
**Date** that specifies when the enhancement expires. All enhancements should have an expiration

date. This date enables the Loadstub component to delete obsolete enhancement data from the Guide database.

If the enhancement is for a current show, this value is not specified. In this case, **EnhLoaderProxy** calculates a fixed expiration date using the date received and show length information.

*dateReceived*
**Date** that specifies the time the enhancement announcement was received. This value is not used for future enhancements.

*lShowLength*
**Long** that specifies the number of minutes until the show ends. This value does not need to be exact, as long as it is equal to or greater than the number of minutes left in the show. This value is not used for future enhancements.

*bCurrentShow*
**Boolean** that indicates whether this is an announcement for an enhanced show that is currently being broadcast. This **Boolean** can be one of the following values.

| Value | Meaning |
|---|---|
| **True** | Current enhancement. In this case, **EnhLoaderProxy** calculates the expiration date and, if necessary, gets the show reference and preload URL data from **EnhCtrl** before loading the data into the Guide database. |
| **False** | Future enhancement. In this case, **EnhLoaderProxy** loads the data into the Guide database. |

**Remarks**

For current enhancements, this method calculates the ending time. If the show reference and preload URL values are not specified, **LoadEnhancement** uses the show reference and preload URL of the current show. It gets these values by querying the enhancement control, **EnhCtrl**.

This method is essentially a wrapper for **ITelevisionServices::LoadEnhancement**. Once the enhancement data is resolved, **EnhLoaderProxy.LoadEnhancement** calls **ITelevisionServices::LoadEnhancement** to load the data into the Guide database.

## EnhLoaderProxy.SetMulticastIP

[This is preliminary documentation and subject to change.]

The SetMulticastIP event occurs when **EnhLoaderProxy** receives an enhancement announcement for the currently displayed show. **EnhLoaderProxy** tunes the enhancement control to the IP address specified in the event. This enables the enhancement control to immediately start receiving triggers.

### Syntax

```
Private Sub object_SetMulticastIP(bstrNetcard As String, _
 bstrAddress As String, iPort As Integer)
```

### Parts

*object*
  Object expression that resolves to the **EnhLoaderProxy** object.
*bstrNetcard*
  **String** that contains the network card address. This address should be in the format *xxx.xxx.xxx.xxx,* for example 125.125.125.125.
*bstrAddress*
  **String** that contains the IP address specified in the matching announcement.
*iPort*
  Integer that specifies the IP port specified in the matching announcement.

## Stream Compiler Object Library

[This is preliminary documentation and subject to change.]

The stream compiler object library, Stream.dll, is a component that loads, stores, and edits enhancement streams. The stream can then be transmitted to the client as an IP stream by the enhancement listener control.

**Note** The stream compiler object library, Stream.dll, is not part of the software supporting the Broadcast Architecture Programmer's Reference. To locate this library, see Further Information on Data Services for the Client.

The stream compiler exposes the following COM objects. Using these, you can write a custom stream editor or an enhancement stream player.

| Object | Description |
|---|---|
| **Event** | An enhancement event object. |
| **Events** | A collection of enhancement event objects. |

### Event

[This is preliminary documentation and subject to change.]

The **Event** object stores information about an enhancement stream event.

| Property | Description |
|---|---|
| **Before** | A value that indicates whether the compiler should download all dependencies of the event before the event's start time. |
| **Handle** | A unique identifier for the event. This property is read-only. |
| **HRef** | Indicates whether HTML hyperlinks should be treated as dependencies for this event. |
| **IsAnnounce** | Indicates whether the event is an enhancement announcement. |
| **IsTrigger** | Indicates whether the event is a trigger. |
| **Length** | Length of the file, in bytes. This property is read-only. |
| **Name** | Name or text value of the event. |
| **onefile** | Indicates whether to send the file and all its dependencies as a single FTS file transmission. |
| **Only** | Indicates whether dependencies, such as images and animations, should be resolved for this event. |
| **Priority** | FTS priority value.<br><br>This property is not yet implemented. |
| **Repeat** | The interval, specified in seconds, between repetitions of the event. |
| **Start** | Start time of the event, in seconds. |
| **Text** | Stream language text of the event. |
| **Timeout** | Time at which the event is garbage collected.<br><br>This method is not implemented in version 1.0 of Broadcast Architecture. |
| **Trigger** | Numerical identifier that identifies the trigger type. |
| **Until** | Duration of the event repetition, in seconds. |
| **XMitDuration** | Time that it will take to transmit the event. This value is read-only. |
| **XMitName** | Filename of the file on the client. In other words, the destination filename. This property is read-only. |

| Method | Description |
|---|---|
| **Delete** | Removes the event from the **Events** collection. |

**Note** The stream compiler object library supplying the **Event** object class, Stream.dll, is not part of the software supporting the Broadcast Architecture Programmer's Reference. To locate this library, see Further Information on Data Services for the Client.

## Event.XMitDuration

[This is preliminary documentation and subject to change.]

The **XMitDuration** property contains the time required to transmit the event. This property is read-only.

### Syntax

```
[ dDuration = ] object.XMitDuration
```

### Parts

*object*
Object expression that resolves to an **Event** object.
*lDuration*
**Double** that receives the number of seconds required to transmit the event.

### Remarks

**XmitDuration** is calculated based on the redundancy, length, and available bandwidth. The following equation is used to calculate the time required to transmit the event:

XmitDuration = ((1 + 1/Redundancy) * Length ) / (Bandwidth/8)

### See Also

**Event.XMitName**, **Event.Name**, **Event.Length**, **Events.Overhead**, **Events.Redundancy**, **Events.Bandwidth**

## Event.onefile

[This is preliminary documentation and subject to change.]

The **onefile** property indicates whether the enhancement file and its dependencies should be transmitted as a single file.

### Syntax

```
object.onefile [ = lfile ]
```

### Parts

*object*
: Object expression that resolves to an **Event** object.

*lfile*
: **Long** that indicates whether to transmit the file and its dependencies as a single file. If this value is non-zero, the files are transmitted as a single file. If this value is zero, they are not.

### Remarks

Transmitting the enhancement file and its dependencies as a single file saves bandwidth. This is because the FTS protocol imposes a high overhead, 5 KB per file, for each file it transmits. Packing the files together reduces the amount of wasted bandwidth.

When **onefile** is set to a non-zero value, the stream compiler resolves the enhancement file's dependencies, and creates a CAB file that packages the file with its dependencies. This CAB file is transmitted to the client and unpackaged by the enhancement control, **EnhCtrl**.

### See Also

Dependencies, **Event.Only**

## Event.IsAnnounce

[This is preliminary documentation and subject to change.]

The **IsAnnounce** property indicates whether the event is an enhancement announcement.

### Syntax

```
object.IsAnnounce [ = lAnnc ]
```

### Parts

*object*
: Object expression that resolves to an **Event** object.

*lAnnc*
: **Long** that indicates whether the event is an announcement. If this value is non-zero, the event is

an announcement. If it is zero, the event is not an announcement.

**See Also**

**Event.IsTrigger**, **Event.Name**

# Event.HRef

[This is preliminary documentation and subject to change.]

The **HRef** property indicates when HTML hyperlink dependencies should be transmitted.

**Syntax**

```
object.HRef [ = lhref ]
```

**Parts**

*object*
Object expression that resolves to an **Event** object.

*lhref*
**Long** that specifies when dependencies should be resolved. This can a value in one of the following ranges:

| Value | Description |
|---|---|
| < 0 | Transmit hyperlink dependencies before the event. |
| 0 | Do not transmit hyperlink dependencies. |
| > 0 | Transmit hyperlink dependencies after the event. Values of *lhref* greater than 0 are not supported in version 1.0 of Broadcast Architecture. |

**Remarks**

If this property is not set, the value in **Events.HRef** is used.

If **Events.HRef** value is also not set, the default is 0, which causes the compiler not to schedule transmission of hyperlink dependencies.

**See Also**

Dependencies, **Events.HRef**

# Event.XMitName

[This is preliminary documentation and subject to change.]

The **XMitName** property indicates the name of the transmitted file at the destination. This property is read-only.

### Syntax

```
[ sName = ] object.XMitName
```

### Parts

*object*
    Object expression that resolves to an **Event** object.
*sName*
    **String** that receives the filename.

### Remarks

This is value that **ipsend** should use as the *sDstFilename* parameter of the **SendFTSFile** method.

The value of **XmitName** is created from the values of **Events.ShowName** and the filename of the transmitted file as follows:

XmitName = "ShowName/FileName"

For example, if the title of the show's enhancements is "My Enhanced Show" and the file being transmitted is "File1.htm" the value of **Event.XmitName** is "My Enhanced Show/File1.htm"

### See Also

**Event.XMitDuration**, **Events.ShowName**

# Event.Handle

[This is preliminary documentation and subject to change.]

The **Handle** property is a unique identifier for the event. This property is read-only.

### Syntax

**[** *lhandle* **= ]** *object*.**Handle**

### Parts

*object*
Object expression that resolves to an **Event** object.
*lhandle*
**Long** that uniquely identifies the event.

### Remarks

**Events.FindHandle** searches the **Events** collection to return the event with the specified handle.

### Examples

The following example uses **Event.Handle** to save a handle to an event in the enhancement stream, and then uses **Events.FindHandle** to recall the event.

```
'Declare variables
Dim Ev As IEvents
Dim e As IEvent
Dim lHandle As Long

'create the collection
Set Ev = New Events

'add an event to the collection
Call Ev.Add(15, "EventA")

'save a handle to EventA,
'(which is currently the last-added event)
lHandle = Ev.LastAdd.Handle

...

'recall EventA from the collection using the previously-saved handle
Set e = Ev.FindHandle(lHandle)
```

## Event.Text

[This is preliminary documentation and subject to change.]

The **Text** property contains the stream language text for the event.

### Syntax

```
object.Text [ = sText ]
```

### Parts

*object*
: Object expression that resolves to an **Event** object.

*sText*
: **String** that contains the stream language text of the event. To locate more information about stream compiler language syntax, see Further Information on Data Services for the Client.

### See Also

**Events.AddText**

## Event.Trigger

[This is preliminary documentation and subject to change.]

The **Trigger** property contains an identifier that specifies the trigger type of this event. This property is only valid if **Event.IsTrigger** is non-zero.

### Syntax

```
object.Trigger [ = lTrigger ]
```

### Parts

*object*
: Object expression that resolves to an **Event** object.

*lTrigger*
: **Long** that specifies the type of trigger. For more information about trigger identifiers and format, see Enhancement Triggers.

### See Also

**Event.IsTrigger**

# Event.Before

[This is preliminary documentation and subject to change.]

The **Before** property indicates whether the compiler should ensure that all dependencies of the event are transmitted before the event's start time, as specified in **Event.Start**.

Note that verifying the event's readiness can include transmitting HTML hyperlink dependencies if **Href** is non-zero.

### Syntax

```
object.Before [ = lBefore ]
```

### Parts

*object*
: Object expression that resolves to an **Event** object.

*lBefore*
: **Long** that indicates whether the event should be verified before it is fired. If this value is non-zero, the event should be verified. If it is zero, it should not.

### See Also

Dependencies

# Event.Timeout

[This is preliminary documentation and subject to change.]

The **Timeout** property indicates the number of seconds after the **Event.Start** time that the event should be deleted by the client application. This value overrides the value set in **Events.TimeOut**.

This property is not implemented in version 1.0 of Broadcast Architecture.

### Syntax

```
object.Timeout [ = dTimeout ]
```

### Parts

*object*
: Object expression that resolves to an **Event** object.

*dTimeout*
: **Double** that indicates the number of seconds after the start time that the event should be deleted. If this value is not set, the default is 0.

### Remarks

If no value is specified for this property, the default value for the enhancement stream, specified in **Events.TimeOut**, is used.

### See Also

**Event.Priority**, **Events.Priority**

## Event.Priority

[This is preliminary documentation and subject to change.]

The **Priority** property specifies the FTS priority.

This property is not implemented in version 1.0 of Broadcast Architecture.

### Syntax

```
object.Priority [ = lPriority ]
```

### Parts

*object*
: Object expression that resolves to an **Event** object.

*lPriority*
: **Long** that specifies the priority. If this value is not set, the default is 5.

### Remarks

The compiler uses the priority value during optimization, when it fills available bandwidth. This value is also used during an interleaved transmission of multiple files. The file with greater weight will be proportionally transmitted more often.

### See Also

**Events.Priority**

# Event.Length

[This is preliminary documentation and subject to change.]

The **Length** property specifies the length of the file, in bytes. This property is read-only.

**Syntax**

```
[ lLength = ] object.Length
```

**Parts**

*object*
Object expression that resolves to an **Event** object.
*lLength*
Long that receives the file length, in bytes. If the file cannot be opened, either because the file does not exist or because you do not have permissions, the length is 0.

**See Also**

**Event.XmitDuration**

# Event.Until

[This is preliminary documentation and subject to change.]

The **Until** property specifies the end time for a repeated event, in seconds. In other words, unless **Event.Repeat** is zero, the event will repeat for **Until** seconds.

**Syntax**

```
object.Until [ = dUntil ]
```

**Parts**

*object*
Object expression that resolves to an **Event** object.
*dUntil*

**Double** that specifies the number of seconds.

### Remarks

The value of **Until** is ignored if **Event.Repeat** is zero.

If **Repeat** is nonzero and **Until** is not specified or is set to zero, the repetition continues for the duration of the show, as specified in **Events.ShowLength**.

## Event.Repeat

[This is preliminary documentation and subject to change.]

The **Repeat** property specifies the number of seconds between repetitions of the event.

### Syntax

```
object.Repeat [ = dRepeat ]
```

### Parts

*object*
Object expression that resolves to an **Event** object.

*dRepeat*
**Double** that specifies the repeat interval, in seconds. A value of 0, indicates that the event should not be repeated. If this value is not zero, it must be greater than 0.1 seconds.

Values less than zero are not supported.

### Remarks

Events are typically transmitted repeatedly so that if a user misses an enhancement transmission due to line noise or because they tuned to another channel, they will be able to receive a subsequent transmission.

### See Also

**Event.Until**

## Event.IsTrigger

[This is preliminary documentation and subject to change.]

The **IsTrigger** property indicates whether the event is a trigger.

### Syntax

*object*.**IsTrigger [ =** *lTrig* **]**

### Parts

*object*
Object expression that resolves to an **Event** object.

*lTrig*
**Long** that indicates whether the event is a trigger. If the value is non-zero, the event is a trigger. If the value is zero, it is not.

### See Also

**Event.IsAnnounce**, **Event.Trigger**

### Examples

The following example searches through the enhancement stream until it finds a trigger event. When it finds a trigger, it stores the trigger's handle in an array.

```
Dim num As Integer
Dim TrigArray() As Long
ReDim TrigArray(0 To evs.Count - 1)

For i = 0 to evs.Count-1
    If (evs.Item(i).IsTrigger) Then
        TrigArray[num] = evs.Item(i).Handle
        num = num + 1
    End If
Next i
```

The reason that the handle is stored instead of the trigger's index value, is that the handle uniquely identifies the trigger and does not change. Because the collection is sorted by start time, the trigger's index value can change as events are added or deleted.

## Event.Only

[This is preliminary documentation and subject to change.]

The **Only** property indicates whether the client application should transmit dependencies for the event.

### Syntax

```
object.Only [ = lOnly ]
```

### Parts

*object*
 Object expression that resolves to an **Event** object.
*lOnly*
 **Long** that indicates whether dependencies should be transmitted for the event. If this value is non-zero, dependencies should be not be transmitted. If it is zero, they should be.

### Remarks

You set the **Only** property to non-zero to prevent the stream compiler from scheduling transmission of dependent files such as GIFs and FutureSplash animations. This is useful, for example, in cases where the dependencies already reside on the client machine, either transmitted by some other means such as webcasting, a previous event, or installed explicitly by the user.

## Event.Name

[This is preliminary documentation and subject to change.]

The **Name** property specifies the name or text of the event. An example of using this property to store event text is a stock ticker enhancement where **Name** stores the stock name and price.

### Syntax

```
object.Name [ = sName ]
```

### Parts

*object*
 Object expression that resolves to an **Event** object.
*sName*
 **String** that contains the name of the event.

### Remarks

The default content of **Name** varies with the type of event. See the following table for specifics.

| **Event Type** | **Content** |
|---|---|
| Announcement | The path and filename of the file that specifies the announcement. |
| Trigger | The trigger data. |
| FTS data | The source filename. |

# Event.Start

[This is preliminary documentation and subject to change.]

The **Start** property specifies the time of the event, in seconds. This is specified in relation to the start time of the episode that the event enhances.

### Syntax

```
object.Start [ = dStart ]
```

### Parts

*object*
    Object expression that resolves to an **Event** object.
*dStart*
    **Double** that specifies the start time of the event. This is expressed as the number of seconds after the episode's start time.

### Remarks

Typically, the time specified by **StartTime** is the starting time of the event. However for some events, such as those where the stream syntax stored in **Event.Text** is of the form **before** *time package***;** this time is the end time of the event.

### See Also

**Event.Text**

# Event.Delete

[This is preliminary documentation and subject to change.]

The **Delete** method removes an event from the **Events** collection.

### Syntax

```
object.Delete()
```

### Parts

*object*
    Object expression that resolves to an **Event** object.

### Remarks

The **Delete** method deletes the instance of the **Event** object that it removes from the collection.

## Events

[This is preliminary documentation and subject to change.]

The **Events** object is a collection of **Event** objects. Using the methods and properties of the **Events** interface you can create or parse an enhancement stream.

The **Events** object has the following properties and methods:

| Property | Description |
|---|---|
| **Bandwidth** | Bandwidth of the transmission medium, in bits per second. |
| **Count** | Number of **Event** objects in the collection.<br><br>This property is read-only. |
| **DependLength** | Length of a file and its dependencies, in bytes. |
| **ErrorCount** | Number of syntax errors in the enhancement stream. |
| **ErrorList** | Retrieves a syntax error from the error array. |
| **FindHandle** | Returns the event associated with the specified handle. |
| **FindTime** | Returns the event starting at the specified time. |
| **FPS** | Frames per second of the enhancement stream. |
| **FrameName** | Retrieves a name from frame name array. |

| | |
|---|---|
| **HRef** | Indicates when HTML hyperlink dependencies should be resolved. |
| **Item** | Retrieves the specified event from the collection. |
| **LastAdd** | Retrieves the event last added to the collection. |
| **LeadTime** | Specifies the amount of time, in seconds, to wait between downloading the last dependency of an trigger event before firing the trigger. |
| **Overhead** | FTS overhead for file transmission, in bytes. |
| **ParseTime** | Converts an extended SMPTE time string into a **Double**. |
| **Priority** | Default transmission priority for the enhancement stream. This property is not yet implemented. |
| **Redundancy** | FTS redundancy, stored as the reciprocal of the extra duration. |
| **ShowLength** | Length of the show associated with this enhancement stream, in seconds. |
| **ShowName** | Name of the show or episode associated with the enhancement stream. |
| **Style** | Reserved. This property is not yet implemented. |
| **TimeOut** | Default time-out interval for the enhancement stream. |
| **TimeStr** | Converts a **Double** into an extended SMPTE time string. |
| **Title** | Title of the enhancement stream. |

| Method | Description |
|---|---|
| **Add** | Add an event object to the enhancement stream. |
| **AddText** | Add an event to the enhancement stream by specifying stream language text. |
| **Clear** | Remove all objects from the enhancement stream. |
| **Flatten** | Convert high-level events into low-level events. This includes such functionality as scheduling transmission of dependencies and building repetition cycles. |
| **Load** | Load an enhancement stream from a file. |
| **Store** | Save the enhancement stream to a file. |

**Note** The stream compiler object library supplying the **Event** object class, Stream.dll, is not part of the software supporting the Broadcast Architecture Programmer's Reference. To locate this library, see Further Information on Data Services for the Client.

# Events.Title

[This is preliminary documentation and subject to change.]

The **Title** property contains the title of the enhancement stream.

### Syntax

```
object.Title [ = sTitle ]
```

### Parts

*object*
    Object expression that resolves to an **Events** object.
*sTitle*
    **String** that contains the title. If this value is not set, the default is an empty string ("").

### Remarks

**Title** specifies the title of the enhancement stream, not the enhanced episode. The title of the enhanced episode is stored in **Events.ShowName**.

# Events.Redundancy

[This is preliminary documentation and subject to change.]

The **Redundancy** property contains the FTS redundancy.

### Syntax

```
object.Redundancy [ = lRedun ]
```

### Parts

*object*
    Object expression that resolves to an **Events** object.
*lRedun*
    **Long** that indicates the FTS redundancy. The default value is 5.

### Remarks

The value of **Redundancy** is the reciprocal of the amount of extra bandwidth used for *forward error correction*. For example, if you set the redundancy to 10, the amount of bandwidth allotted to forward error correction is 1/10, or 10%. In other words, an additional number of bytes, equal to 10% of the original file, will be transmitted for error correction.

### See Also

**Events.Overhead**, **Event.XmitDuration**

## Events.Overhead

[This is preliminary documentation and subject to change.]

The **Overhead** property specifies the fixed overhead for FTS transmission, in bytes.

### Syntax

```
object.Overhead [ = lOverhead ]
```

### Parts

*object*
 Object expression that resolves to an **Events** object.
*lOverhead*
 **Long** that contains the FTS overhead, in bytes. The default value is 5120.

### See Also

**Event.Length**, **Events.Redundancy**, **Event.XmitDuration**

## Events.Style

[This is preliminary documentation and subject to change.]

The **Style** property is reserved for use in future versions of Broadcast Architecture.

### Syntax

```
object.Style [ = lStyle ]
```

### Parts

*object*
: Object expression that resolves to an **Events** object.

*lStyle*
: **Long**. The value default is 0.

### Remarks

This property is not currently used.

## Events.HRef

[This is preliminary documentation and subject to change.]

The **HRef** property indicates when HTML hyperlink dependencies should be resolved.

### Syntax

```
object.HRef [ = lHref ]
```

### Parts

*object*
: Object expression that resolves to an **Events** object.

*lhref*
: **Long** that specifies when dependencies should be resolved. This can be a value in one of the following ranges:

| Value | Description |
|---|---|
| < 0 | Resolve dependencies before the event. |
| 0 | Do not resolve dependencies. |
| > 0 | Resolve dependencies after the event.<br><br>Values of *lhref* greater than 0 are not supported in version 1.0 of Broadcast Architecture. |

### Remarks

This value can be overridden by individual events by setting the **Event.HRef** property. If neither value is set, the default is 0.

## Events.ShowName

[This is preliminary documentation and subject to change.]

The **ShowName** property is the name of the show or episode associated with the event stream.

**Syntax**

```
object.ShowName [ = sName ]
```

**Parts**

*object*
: Object expression that resolves to an **Events** object.

*sName*
: **String** that contains the name of the show. If this value is not set, the default is an empty string ("").

**Remarks**

This property is used to compute **Event.XmitName**.

**See Also**

**Events.Title**

## Events.FPS

[This is preliminary documentation and subject to change.]

The **FPS** property retrieves or sets the frames per second.

**Syntax**

```
object.FPS [ = dfps ]
```

**Parts**

*object*
    Object expression that resolves to an **Events** object.
*dfps*
    **Double** that indicates the frames per second.

**See Also**

**Events.ParseTime**

## Events.FrameName

[This is preliminary documentation and subject to change.]

The **FrameName** property retrieves or sets the names of HTML frames. These values are stored in an array.

**Syntax**

```
object.FrameName( lIndex ) [ = sName ]
```

**Parts**

*object*
    Object expression that resolves to an **Events** object.
*lIndex*
    **Long** that indicates which element of the 0-based array to set or retrieve.
*sName*
    **String** that contains the name of the HTML frame.

**Remarks**

For example, in a multiframe enhancement, you can use **FrameName** to store the names of the enhancement frames. Your application can then retrieve the name of a particular frame from the array, instead of explicitly referencing the frames by name.

Using the **FrameName** property instead of explicit frame names makes your application more portable and easier to maintain. If the names of the enhancement frames change, you will only have to update them once.

**Examples**

The following example uses **FrameName** to store the names of three HTML frames names. It then retrieves the name of the first frame to add a NavBase trigger event.

```
'store the frame names in the array
evs.FrameName(0) = "Left"
evs.FrameName(1) = "Video"
evs.FrameName(2) = "Bottom"

evs.AddText("00:10:30:00 trigger ( 3 "+evs.FrameName(0)+" ""NewEnh.htm"") only;")
```

# Events.TimeStr

[This is preliminary documentation and subject to change.]

The **TimeStr** method converts a double into a SMPTE time string. This property is read-only.

### Syntax

```
[ sTime = ] object.TimeStr( dTime )
```

### Parts

*object*
    Object expression that resolves to an **Events** object.
*dTime*
    **Double** that specifies the time.
*sTime*
    **String** that receives the time specified in *dTime* as an SMPTE time string, of the format "hh:mm:ss.dd".

### Examples

The following example sets the variable `sTime` to "00:05:42.00".

```
Dim sTime As String
Dim evs As Events
Set evs = new Events

sTime = evs.TimeStr(342.0)
```

### See Also

**Events.ParseTime**

# Events.ParseTime

[This is preliminary documentation and subject to change.]

The **ParseTime** method converts a SMPTE time string into a **Double**. This property is read-only.

### Syntax

```
[ dTime = ] object.ParseTime( sTime )
```

### Parts

*object*

Object expression that resolves to an **Events** object.

*sTime*

**String** that specifies the time as an SMPTE time string. This can be one of the following formats, depending on the current frame rate.

| SMPTE time format | Format | Example |
|---|---|---|
| 30 frames per second | hh:mm:ss:ff | 01:00:30:15 |
| drop frame (29.97 frames per second) | hh:mm:ss;ff | 01:00:30;15 |
| fractional seconds | hh:mm:ss.dd | 01:00:30.5 |

*dTime*

**Double** that receives the time specified in *sTime* in numerical format.

### Remarks

You can set the frame rate using **Events.FPS**.

### Examples

The following example sets the variable `dTime` to 342.0.

```
Dim dTime As Double
Dim evs As Events
Set evs = new Events

dTime = evs.ParseTime("00:05:42:00")
```

### See Also

**Events.TimeStr**

# Events.DependLength

[This is preliminary documentation and subject to change.]

The **DependLength** property contains the length of a file and its dependencies, in bytes. This property is read-only.

### Syntax

```
[ lLength = ] object.DependLength( sFilename )
```

### Parts

*object*
: Object expression that resolves to an **Events** object.

*sFilename*
: **String** that contains the name of the file.

*lLength*
: **Long** that receives the total number of bytes in the file and all its dependencies.

### See Also

Dependencies, **Event.Length**, **Event.XmitDuration**

# Events.FindHandle

[This is preliminary documentation and subject to change.]

The **FindHandle** property returns a reference to the event that has the specified handle. This property is read-only.

### Syntax

```
Set oEvent = object.FindHandle ( lHandle )
```

### Parts

*object*
: Object expression that resolves to an **Events** object.

*lHandle*

**Long** that contains the event handle. This value will be compared to each event's **Event.Handle** property until a match is found. If a match is not found, **FindHandle** returns an error

*oEvent*

**Event** that receives the event specified by *lHandle*.

### Examples

The following example saves a handle to an event in the enhancement stream, and then uses **FindHandle** to recall that event.

```
'Declare variables
Dim Ev As IEvents
Dim e As IEvent
Dim lHandle As Long

'create the collection
Set Ev = New Events

'add an event to the collection
Call Ev.Add(15, "EventA")

'save a handle to EventA,
'(which is currently the last-added event)
lHandle = Ev.LastAdd.Handle

...

'recall EventA from the collection using the previously-saved handle
Set e = Ev.FindHandle(lHandle)
```

### See Also

**Events.FindTime**

## Events.LeadTime

[This is preliminary documentation and subject to change.]

The **LeadTime** property specifies the time to wait after downloading a trigger event's dependency files before firing the trigger.

### Syntax

*object*.**LeadTime** [ = *dTime* ]

### Parts

*object*
  Object expression that resolves to an **Events** object.

*dTime*
  **Double** that specifies the number of seconds the transmitter should wait. This must be a positive value. The default value is 5.

### See Also

Dependencies

## Events.FindTime

[This is preliminary documentation and subject to change.]

The **FindTime** property returns a reference to the event that starts at the specified time. This property is read-only.

### Syntax

```
Set oEvent = object.FindTime( dTime )
```

### Parts

*object*
  Object expression that resolves to an **Events** object.

*dTime*
  **Double** that specifies the start time. This value will be compared to each events **Event.Start** property until a match is found. If a match is not found, **FindTime** returns an error.

*oEvent*
  **Event** object that receives the reference to the event with the start time specified by *dTime*.

### Examples

The following example looks up an event in the **Events** collection by start time.

```
'Declare variables
Dim Ev As IEvents
Dim e As IEvent
Dim lHandle As Long

'create the collection
Set Ev = New Events

'add events to the collection
Call Ev.Add(15, "EventA")
Call Ev.Add(45, "EventB")
Call Ev.Add(25, "EventC")

'find EventB in the collection by specifying
'its start time.
Set e = Ev.FindTime(45)
```

# Events.AddText

[This is preliminary documentation and subject to change.]

The **AddText** method adds an event to the enhancement stream using the stream language.

### Syntax

*object*.**AddText(** *sText* **)**

### Parts

*object*
: Object expression that resolves to an **Events** object.

*sText*
: **String** that contains the stream language. This value is used to initialize the **Event.Text** property of the new event. If this string is empty (""), the event is not added.

  To locate more information about stream compiler language syntax, see Further Information on Data Services for the Client.

### Remarks

You can also use this method to set the value of enhancement stream global variables, namely BandWidth, Href, FrameName, Overhead, Priority, FPS, Redundancy, TimeOut, LeadTime, Style, ShowLength, and ShowName. The value of these global variables are wrapped by the **Events** properties of the same name.

For example, you can set the value of **Events.Redundancy** to 10 with the following:

```
evs.AddText("Redundancy = 10;")
```

### Examples

The following example adds a NavBase trigger event to the enhancement stream. Note that because the **only** keyword is specified, the file NewEnh.htm is assumed to already exist on the client's computer and is not scheduled to be transmitted.

```
evs.AddText("00:10:30:00 trigger ( 3 ""Left"" ""NewEnh.htm"") only;")
```

### See Also

**Events.Add**

## Events.Flatten

[This is preliminary documentation and subject to change.]

The **Flatten** method converts high-level enhancement stream events into low-level ones. This method can cause the stream compiler to add new events and change event times slightly.

It is recommended that your application call **Flatten** before it transmits the enhancement stream.

### Syntax

```
object.Flatten()
```

### Parts

*object*
Object expression that resolves to an **Events** object.

### Remarks

The **Flatten** method resolves high-level events into low-level ones. For example, if you had defined an FTS-download event for an HTML file that contained four GIFs, the **Flatten** method would turn this event into five FTS-download events, one for the main file and each image dependency.

## Events.LastAdd

[This is preliminary documentation and subject to change.]

The **LastAdd** property returns a reference to the event that was last added to the enhancement stream. This property is read-only.

### Syntax

```
[ Set oEvent = ] object.LastAdd
```

### Parts

*object*

Object expression that resolves to an **Events** object.

*oEvent*

**Event** that receives the last event added to the collection.

### Examples

The following example displays a message box containing the text Second Event, because that was the event most recently added to the collection.

```
'Declare variables
Dim Ev As IEvents
Dim e As IEvent

'Create the Events collection
Set Ev = New Events

'Add two events to the collection
Call Ev.Add(35, "First Event")
Call Ev.Add(25, "Second Event")

'retrieve the last-added event
Set e = Ev.LastAdd

'display the name of the last-added event
MsgBox e.Name
```

### See Also

**Event.Handle**, **Events.Add**, **Events.AddText**, **Events.FindHandle**, **Events.FindTime**

# Events.ErrorCount

[This is preliminary documentation and subject to change.]

The **ErrorCount** property contains a count of the syntax errors in the enhancement stream. It counts the syntax errors discovered in the last call to **Events.Load** or **Events.AddText**. This property is read-only.

### Syntax

```
[ lCount = ] object.ErrorCount
```

### Parts

*lCount*
    **Long** that receives the number of syntax errors.
*object*
    Object expression that resolves to an **Events** object.

### Remarks

The **Events.ErrorList** property contains an array of strings that describe the syntax errors. The error information includes the line number that contains the syntax error. The line number specifies the line in the enhancement stream file loaded by **Events.Load**.

### Examples

The following example displays a message box indicating the number of syntax errors currently in the enhancement stream.

```
Msgbox "Number of syntax errors:"+CStr(evs.ErrorCount)
```

## Events.Errorlist

[This is preliminary documentation and subject to change.]

The **Errorlist** property retrieves a specific error from the syntax error array created during a call to **Events.Load** or **Events.AddText**. This property is read-only.

### Syntax

```
[ sList = ] object.Errorlist( lIndex )
```

### Parts

*sList*
    **String** that describes the syntax error.
*object*
    Object expression that resolves to an **Events** object.
*lIndex*
    **Long** that indicates which element of the array to return.

### Remarks

The **ErrorList** property contains an array of strings that describe the syntax errors. The error information includes the line number that contains the syntax error. The line number specifies the line in the enhancement stream file loaded by **Events.Load**.

To fix a syntax error, you would typically open and edit the stream syntax file in a text error such as Notepad.

**See Also**

**Events.ErrorCount**

## Events.Priority

[This is preliminary documentation and subject to change.]

The **Priority** property indicates the default priority for the events in the enhancement stream. This property is not yet implemented.

**Syntax**

```
object.Priority [ = lPriority ]
```

**Parts**

*object*
Object expression that resolves to an **Events** object.
*lPriority*
**Long** that indicates the default priority.

**Remarks**

This property can be overridden for an individual event by setting **Event.Priority**. If neither value is set, the default priority is 5.

## Events.TimeOut

[This is preliminary documentation and subject to change.]

The **TimeOut** property indicates the default number of seconds after which events are deleted from the enhancement spool directory.

### Syntax

```
object.TimeOut [ = dTimeout ]
```

### Parts

*object*
 Object expression that resolves to an **Events** object.
*dTimeout*
 **Double** that indicates the number of seconds for garbage collection. The default value is 0.

### Remarks

You can override this value for individual events by setting the **Event.Timeout** property.

The deletion of expired enhancements is automatically handled by the enhancement control, **EnhCtrl**, on the client computer.

### See Also

**Events.ShowLength**

## Events.ShowLength

[This is preliminary documentation and subject to change.]

The **ShowLength** property specifies the length of the enhanced episode, in seconds.

### Syntax

```
object.ShowLength [ = dLength ]
```

### Parts

*object*
 Object expression that resolves to an **Events** object.
*dLength*
 **Double** that indicates the show length, in seconds. The default value is 0.

### See Also

**Event.Timeout**

## Events.Bandwidth

[This is preliminary documentation and subject to change.]

The **Bandwidth** property indicates the bandwidth of the transmission medium.

### Syntax

```
object.Bandwidth [ = dBps ]
```

### Parts

*object*
Object expression that resolves to an **Events** object.

*dBps*
**Double** that specifies the transmission rate, in bits per second. If this value is not set, the default is 9600.

### Remarks

This value is used by the stream compiler during flattening and optimization of the stream.

### See Also

**Event.Length**, **Events.Overhead**, **Events.Redundancy**, **Event.XMitDuration**

## Events.Add

[This is preliminary documentation and subject to change.]

The **Add** method creates an event and adds it to the enhancement stream.

### Syntax

```
Call object.Add( dTime, sName )
```

### Parts

*object*
: Object expression that resolves to an **Events** object.

*dTime*
: **Double** that contains the start time of the event, in seconds. This value is used to initialize the **Event.Start** property of the new object.

*sName*
: **String** that contains the event name. This value is used to initialize the **Event.Name** property of the new event.

### Remarks

The Visual Basic implementation of **Add** does not return a reference to the newly added event. However, you can retrieve the event from the collection by using the **LastAdd**, **FindTime**, or **FindHandle** properties. Once you have a reference to the object, you can edit its properties or delete it from the collection.

### Examples

The following example creates an Events collection and adds two enhancement events to the collection.

```
'Declare variables
Dim Ev As IEvents
Dim e As IEvent

'Create the Events collection
Set Ev = New Events

'Add two events to the collection
Call Ev.Add(35, "EventA")
Call Ev.Add(25, "EventB")
```

### See Also

**Events.AddText**

## Events.Clear

[This is preliminary documentation and subject to change.]

The **Clear** method removes all events from the enhancement stream. The event objects are destroyed.

### Syntax

```
object.Clear()
```

### Parts

*object*
  Object expression that resolves to an **Events** object.

### Remarks

If you load an enhancement stream from a file without first clearing the current enhancement stream, the two streams are merged.

For example, if you had a stream with events A and B occurring at 15 and 30 seconds, and a second stream with events C and D occurring at 20 and 45 seconds, the merged stream would have events in the following order: A, C, B, and D.

### See Also

**Events.Load**

## Events.Store

[This is preliminary documentation and subject to change.]

The **Store** method saves the enhancement stream to a text file. The stream is stored in stream compiler syntax language.

### Syntax

```
object.Store( sFilename )
```

### Parts

*object*
  Object expression that resolves to an **Events** object.
*sFilename*
  **String** that specifies the filename.

### Remarks

If this file specified by *sFilename* does not currently exist, it is created. If the file exists, it is overwritten.

### Examples

The following example stores the enhancement stream in a file named Stream.txt.

```
evs.Store ("Stream.txt")
```

### See Also

**Events.Load**

## Events.Load

[This is preliminary documentation and subject to change.]

The **Load** method loads an enhancement stream from a file into an **Events** collection.

### Syntax

```
object.Load( sFilename )
```

### Parts

*object*
Object expression that resolves to an **Events** object.

*sFilename*
**String** that specifies the filename.

### Remarks

If there are syntax errors in the stream compiler syntax loaded from the file, these are enumerated in **Events.ErrorCount**. Additional information about the errors is stored in **Events.ErrorList**.

If you load an enhancement stream from a file without first clearing the current enhancement stream, the two streams are merged.

For example, if you had a stream with events A and B occurring at 15 and 30 seconds, and a second stream with events C and D occurring at 20 and 45 seconds, the merged stream would have events in the following order: A, C, B, and D.

### Examples

The following example loads the stream stored in the file Stream.txt into an events collection.

```
evs.Load("Stream.txt")
```

### See Also

**Events.Store**, **Events.Clear**

## Events.Item

[This is preliminary documentation and subject to change.]

The **Item** property returns the specified event from the enhancement stream. This property is read-only.

### Syntax

```
Set oEvent = object.Item( lIndex )
```

### Parts

*object*
Object expression that resolves to an **Events** object.

*lIndex*
**Long** that indicates the 0-based index of the event to retrieve.

### Examples

The following example retrieves the first event in the enhancement stream. Because events are ordered in the collection by start time, this event is the first event that occurs.

```
Set e = evs.Item(0)
```

This could also be written as:

```
Set e = evs(0)
```

Note that because events are stored as a 0-based list, the index of the first item is 0.

### See Also

**Events.Count**

# Events.Count

[This is preliminary documentation and subject to change.]

The **Count** property indicates the number of events in the enhancement stream. This property is read-only.

### Syntax

```
[ lCount = ] object.Count
```

### Parts

*object*
Object expression that resolves to an **Events** object.
*lCount*
**Long** that indicates the number of events.

### Examples

The following example uses the count property to return the last event in the collection.

```
Set e = evs.Item(evs.Count - 1)
```

Note that because events are stored as a 0-based list, the index of the last item is Count – 1.

### See Also

**Events.Item**

# ipsend

[This is preliminary documentation and subject to change.]

The **ipsend** object, provided by IPEnhSnd.dll, is the enhancement listener's counterpart at the head end. This objects transmits the enhancement events, triggers, announcements, and data, over the broadcast medium.

You can call the methods and properties of the **ipsend** object to programmatically control what is sent.

The ipsend object has the following properties and methods.

| Property | Description |
|---|---|
| AnnouncementAddress | IP address on which announcements are sent. |
| AnnouncementPort | IP port on which announcements are sent. |
| FTSAddress | IP address on which FTS transmissions are sent. |
| FTSPort | IP port on which FTS transmissions are sent. |
| NetCard | Specifies the net card used for transmission. |
| TriggerAddress | IP address on which triggers are sent. |
| TriggerPort | IP port on which triggers are sent. |

| Method | Description |
|---|---|
| ConnectAnnouncement | Establishes a connection for transmitting announcements. |
| ConnectFTS | Establishes a connection for transmitting FTS files. |
| ConnectTrigger | Establishes a connection for transmitting triggers. |
| SendAnnouncement | Transmits an announcement. |
| SendDeleteAnnouncement | Transmits a delete announcement. A delete announcement removes enhancement data from the Guide database. |
| SendFTSFile | Transmits an FTS file. |
| SendTrigger | Transmits a trigger. |

**Remarks**

**Note** The stream compiler object library supplying the **ipsend** object class, Stream.dll, is not part of the software supporting the Broadcast Architecture Programmer's Reference. To locate this library, see Further Information on Data Services for the Client.

# ipsend.ConnectTrigger

[This is preliminary documentation and subject to change.]

The **ConnectTrigger** method establishes a connection for sending triggers.

### Syntax

```
object.ConnectTrigger( sAddress, sPort )
```

### Parts

*object*
Object expression that resolves to an **ipsend** object.
*sAddress*
**String** that specifies the IP address. This address should be in the format *xxx***.***xxx***.***xxx***.***xxx*, for example 255.255.255.255.
*sPort*
String that specifies the IP port.

### Remarks

You should call this method before calling **ipsend.SendTrigger**.

**ConnectTrigger** sets the values returned by **ipsend.TriggerAddress** and **ipsend.TriggerPort**.

### See Also

**ipsend.ConnectAnnouncement**, **ipsend.ConnectFTS**

## ipsend.SendTrigger

[This is preliminary documentation and subject to change.]

The **SendTrigger** method sends a trigger over the broadcast medium.

### Syntax

```
object.SendTrigger( lKey, sData )
```

### Parts

*object*
Object expression that resolves to an **ipsend** object.
*lKey*
**Long** that contains the trigger identifier. For more information about trigger identifiers and format, see Enhancement Triggers.
*sData*

**String** that contains the trigger data. The format of the data depends on the type of trigger sent.

### Remarks

Before you call this method, you must call **ipsend.ConnectTrigger** to establish a connection.

### See Also

**ipsend.SendFTSFile**, **ipsend.SendAnnouncement**

## ipsend.TriggerAddress

[This is preliminary documentation and subject to change.]

The **TriggerAddress** property specifies the IP address over which triggers will be sent. This property is read-only.

### Syntax

```
[ sIPAddr = ] object.TriggerAddress
```

### Parts

*object*
Object expression that resolves to an **ipsend** object.

*sIPAddr*
**String** that receives the IP address. This address will be in the format *xxx***.***xxx***.***xxx***.***xxx*, for example 255.255.255.255.

### Remarks

The value returned by **TriggerAddress** is set when you call **ipsend.ConnectTrigger**.

### See Also

**ipsend.TriggerPort**, **ipsend.SendTrigger**

## ipsend.TriggerPort

[This is preliminary documentation and subject to change.]

The **TriggerPort** property specifies the IP port over which triggers are sent. This property is read-only.

### Syntax

```
[ iIPPort = ] object.TriggerPort
```

### Parts

*object*
  Object expression that resolves to an **ipsend** object.
*iIPPort*
  **Integer** that receives the port.

### Remarks

The value returned by **TriggerPort** is set when you call **ipsend.ConnectTrigger**.

### See Also

**ipsend.TriggerAddress**, **ipsend.SendTrigger**

## ipsend.NetCard

[This is preliminary documentation and subject to change.]

The **NetCard** property specifies the network card from which the enhancement stream, including triggers, FTS file transmissions, and announcements, is transmitted.

### Syntax

```
object.NetCard [ = sNetcard ]
```

### Parts

*object*
  Object expression that resolves to an **ipsend** object.
*sNetcard*
  **String** that contains the network card address. This address should be in the format *xxx.xxx.xxx.xxx*, for example 125.125.125.125.

# ipsend.FTSAddress

[This is preliminary documentation and subject to change.]

The **FTSAddress** property specifies the IP address over which FTS transmissions are sent. This property is read-only.

### Syntax

```
[ sIPAddr = ] object.FTSAddress
```

### Parts

*object*
Object expression that resolves to an **ipsend** object.
*sIPAddr*
**String** that receives the IP address. This address will be in the format *xxx***.***xxx***.***xxx***.***xxx*, for example 255.255.255.255.

### Remarks

The value returned by **FTSAddress** is set when you call **ipsend.ConnectFTS**.

### See Also

**ipsend.FTSPort**, **ipsend.SendFTSFile**

# ipsend.FTSPort

[This is preliminary documentation and subject to change.]

The **FTSPort** property specifies the IP port over which FTS transmissions are sent. This property is read-only.

### Syntax

```
[ iIPPort = ] object.FTSPort
```

### Parts

*object*
: Object expression that resolves to an **ipsend** object.

*iIPPort*
: **Integer** that contains the port.

### Remarks

The value returned by **FTSPort** is set when you call **ipsend.ConnectFTS**.

### See Also

**ipsend.FTSAddress**, **ipsend.SendFTSFile**

## ipsend.ConnectFTS

[This is preliminary documentation and subject to change.]

The **ConnectFTS** method establishes a connection for the transmission of FTS data.

### Syntax

```
object.ConnectFTS( sAddress, iPort )
```

### Parts

*object*
: Object expression that resolves to an **ipsend** object.

*sAddress*
: **String** that contains the IP address. This address should be in the format *xxx*.*xxx*.*xxx*.*xxx*, for example 255.255.255.255.

*iPort*
: **Integer** that contains the IP port.

### Remarks

You must call this method before calling **ipsend.SendFTSFile**.

### See Also

**ipsend.ConnectAnnouncement**, **ipsend.ConnectTrigger**, **ipsend.FTSAddress**, **ipsend.FTSPort**

# ipsend.SendFTSFile

[This is preliminary documentation and subject to change.]

The **SendFTSFile** method sends FTS data.

### Syntax

```
object.SendFTSFile( sSrcFilename, sDstFilename )
```

### Parts

*object*
Object expression that resolves to an **ipsend** object.
*sSrcFilename*
**String** that contains the name of the file at the transmission source or head-end.
*sDstFilename*
**String** that contains the name of the file on the client.

### Remarks

Before you call this method, you must call **ipsend.ConnectFTS** to establish a connection.

### See Also

**ipsend.SendAnnouncement**, **ipsend.SendTrigger**, **ipsend.FTSAddress**, **ipsend.FTSPort**, FTSData Trigger

# ipsend.AnnouncementPort

[This is preliminary documentation and subject to change.]

The **AnnouncementPort** property is the IP port to use when sending announcements. This property is read-only.

### Syntax

```
[ iIPPort = ] object.AnnouncementPort
```

### Parts

*object*
: Object expression that resolves to an **ipsend** object.

*iIPPort*
: **Integer** that receives the IP port.

**See Also**

**ipsend.AnnouncementAddress**, **ipsend.ConnectAnnouncement**, **ipsend.SendAnnouncement**

## ipsend.AnnouncementAddress

[This is preliminary documentation and subject to change.]

The **AnnouncementAddress** property is the IP address to use when sending announcements. This property is read-only.

**Syntax**

```
[ sIPAddr = ] object.AnnouncementAddress
```

**Parts**

*object*
: Object expression that resolves to an **ipsend** object.

*sIPAddr*
: **String** that receives the IP address. This address will be in the format *xxx***.***xxx***.***xxx***.***xxx*, for example 255.255.255.255.

**See Also**

**ipsend.AnnouncementPort**, **ipsend.ConnectAnnouncement**, **ipsend.SendAnnouncement**

## ipsend.ConnectAnnouncement

[This is preliminary documentation and subject to change.]

The **ConnectAnnouncement** method establishes a connection over which announcements can be sent.

### Syntax

```
object.ConnectAnnouncement( sIPAddress, iIPPort )
```

### Parts

*object*
Object expression that resolves to an **ipsend** object.
*sIPAddress*
**String** that contains the IP multicast address used to connect to the announcement stream. This address should be in the format *xxx***.***xxx***.***xxx***.***xxx*, for example 255.255.255.255.
*iIPPort*
**String** that contains the IP port.

### Remarks

You should call this method before calling **ipsend.SendAnnouncement**.

### See Also

**ipsend.ConnectFTS**, **ipsend.ConnectTrigger**, **ipsend.AnnouncementAddress**, **ipsend.AnnouncementPort**

## ipsend.SendAnnouncement

[This is preliminary documentation and subject to change.]

The **SendAnnouncement** method sends an enhancement announcement over the broadcast medium.

### Syntax

```
object.SendAnnouncement( sAnncFilename )
```

### Parts

*object*
Object expression that resolves to an **ipsend** object.
*sAnncFilename*
**String** that contains the source filename of the announcement. This file is a text file that specifies the announcement in SDP/SAP protocol. For more information, see Enhancement Announcement Format.

### Remarks

Before you call this method, you must call **ipsend.ConnectAnnouncement** to establish a connection.

### See Also

**ipsend.SendTrigger**, **ipsend.SendFTSFile**, **ipsend.SendDeleteAnnouncement**

# ipsend.SendDeleteAnnouncement

[This is preliminary documentation and subject to change.]

The **SendDeleteAnnouncement** method sends a delete announcement over the broadcast medium. A delete announcement removes data about a specific enhancement from the Guide database.

### Syntax

```
object.SendDeleteAnnouncement( sAnncFilename )
```

### Parts

*object*
Object expression that resolves to an **ipsend** object.

*sAnncFilename*
**String** that contains the source filename of the delete announcement. This file is a text file that specifies the announcement in SDP/SAP protocol. For more information, see Enhancement Announcement Format.

### Remarks

This method deletes data for the enhancement specified in the file, *sAnncFilename*. Note that the content of this file is identical to the content of the file specified in the call to **ipsend.SendAnnouncement**. In other words, you can use the same file to delete enhancement data from the Guide database that you used to set it.

Before you call this method, you must call **ipsend.ConnectAnnouncement** to establish a connection.

### See Also

**ipsend.SendTrigger**, **ipsend.SendFTSFile**, **ipsend.SendAnnouncement**

# Internet Channel Broadcast Client

[This is preliminary documentation and subject to change.]

Internet channel broadcasting is an architecture that enables World Wide Web sites to be collected, packaged, and then broadcast to multiple subscribers simultaneously.

Currently, the Web operates on a one-to-one basis. Each user who wants to view a site must create a separate connection to that site. If too many users try to access a site at the same time, the server is unable to handle all of the requests. In contrast, Internet channel broadcasting operates on a one-to-many basis. Web sites are broadcast to many users at once and stored in each user's cache until the user is ready to view them.

The Internet Channel Broadcast client is a component of the Microsoft® Windows® 98 operating system. This client receives broadcast updates of Web sites that the user has subscribed to and stores the updated files in the Web browser cache. The client extends the Microsoft® Internet Explorer subscription model to a one-to-many broadcast architecture.

For more information, see the following topics:

- About the Internet Channel Broadcast Client, which describes the architecture and functionality of the Internet Channel Broadcast client.
- Using the Internet Channel Broadcast Client, which explains how to use and programmatically extend the Internet Channel Broadcast client.
- Internet Channel Broadcast Client Reference, which details the registry keys and announcement format used by the Internet Channel Broadcast client.

# About the Internet Channel Broadcast Client

[This is preliminary documentation and subject to change.]

The Internet Channel Broadcast client software is part of the Microsoft® Windows® 98 operating system and is installed when you select the Broadcast Data Services component. The client uses an Announcement Listener *filter* to monitor announcements from the Internet Channel Broadcast server. These announcements describe forthcoming Web channel broadcasts.

When the Internet channel broadcast filter receives an announcement about the upcoming broadcast of a Internet channel, it queries Microsoft® Internet Explorer to determine whether the user has subscribed to that channel. If the user has, the filter elects to receive the packaged files. The filter stores received packages in the receiving directory for Internet channel broadcasting, typically C:\Program Files\Webcast\Recv. After a package is received, the filter unpackages it, reconstructs the files, and moves them to the Internet Explorer cache. The user can then view the files in the ordinary fashion, using a Web browser.

For more information, see the following topics:

- Internet Channels
- Subscriptions
- Internet Channel Broadcasting Overview
- Internet Channel Broadcast Filter

## Internet Channels

[This is preliminary documentation and subject to change.]

An Internet channel is a subscribable Web site, in other words a Web site users can subscribe to. A Web site administrator can make a site subscribable by creating a *Channel Definition Format* (CDF) file and publishing it on the Web. The CDF file lists the Web files that make up the channel. These files can be part or all of those in the Web site.

A user subscribes to an Internet channel using Internet Explorer version 4.0. Subscribing enables a user to receive notifications or updates when the channel content changes. For more information on subscribing, see Subscriptions and Subscribing to a Channel.

For more information about Internet Explorer 4.0 and CDF files, see Further Information on Data Services for the Client.

## Subscriptions

[This is preliminary documentation and subject to change.]

Internet channels are subscribable Web sites. When a user subscribes to a channel, the user can elect to receive notifications or updated files when the channel content changes.

Subscriptions are created and managed using Internet Explorer 4.0. When a user subscribes to a channel, Internet Explorer adds an entry to the user's subscriptions folder, C:\Windows\Subscriptions. The entry lists information about the channel and its subscription schedule. Once users have subscribed to a channel, they periodically receive updated files or notifications about changes.

Windows 98 provides two mechanisms to provide the user with updated content from subscribed channels: the Internet Explorer 4.0 model, in which the browser periodically connects to the Web and downloads channel updates, and Internet channel broadcasting, in which channel updates are broadcast to clients.

Internet channel broadcasting provides these advantages:

- Users can receive Web content without being connected to the Internet. For example, if your laptop is equipped with a analog television tuner and a local television station broadcasts Internet channels over the *vertical blanking interval* (VBI), you can receive updates of your favorite Web sites while traveling.
- Web channels are broadcast to clients in a one-to-many transmission. This type of broadcast enables many subscribers to receive Web updates at once without increasing server load. For users, this means that they can view the latest versions of their favorite Web sites immediately, without downloading files over an Internet connection.

For more information about Internet Explorer 4.0, see Further Information on Data Services for the Client.

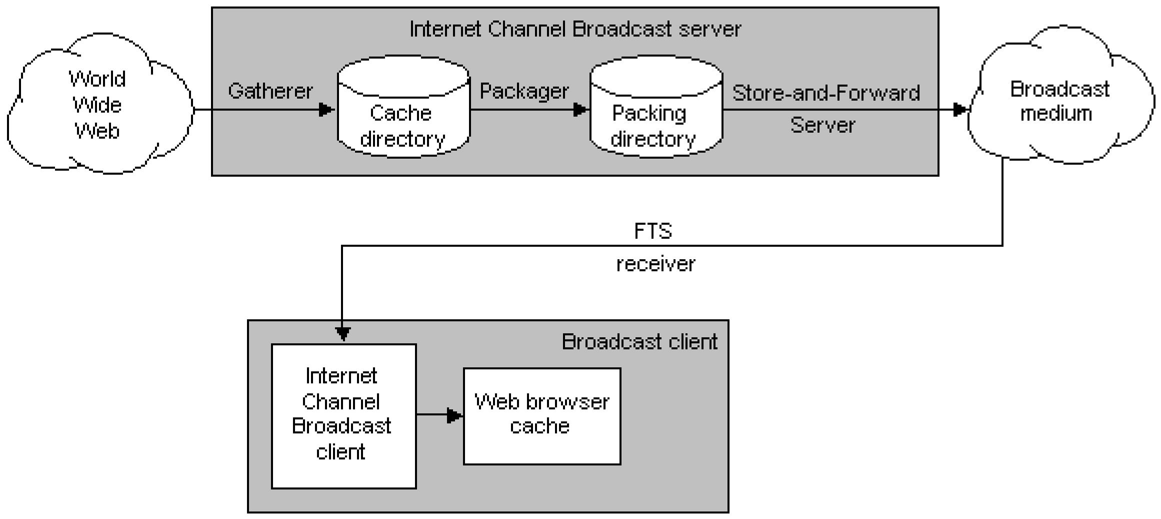
## Internet Channel Broadcasting Overview

[This is preliminary documentation and subject to change.]

Internet channel broadcasting starts at the service provider. The service provider maintains an Internet Channel Broadcast server. This server gathers channels from the Web, packages them, and at scheduled intervals transmits the packaged files over a broadcast medium, such as a local area network (LAN) or analog television signal, to the client.

The Internet Channel Broadcast client monitors announcements, electing to receive updates of channels to which the user is subscribed. Broadcasts of channels the user has not subscribed to are ignored. The client unpackages the updated channel files into the Web browser's cache.

The following diagram illustrates the flow of data through the Internet channel broadcast architecture.



## Internet Channel Broadcast Filter

[This is preliminary documentation and subject to change.]

The Internet Channel Broadcast filter, Webfilt.dll, is the heart of the client application. This filter interacts with other Windows 98 components, such as Internet Explorer and the Microsoft® NetShow™ server, to filter and receive incoming channel updates. This filter is an Announcement Listener *filter* that recognizes and responds to announcements about Internet channels to be broadcast.

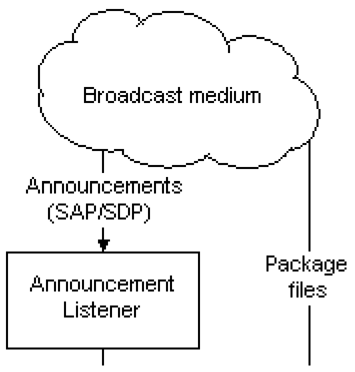
When the filter receives an announcement about an upcoming broadcast of a Internet channel, it queries Internet Explorer to determine whether the user has subscribed to that channel. If the user has, the filter calls the NetShow *File Transfer Service* (FTS) receiver, Nsfts.dll, to receive the packaged files and store them in the receiving directory, by default C:\Program Files\Webcast\Recv. If the user has not subscribed to the channel, the filter ignores the transmission.

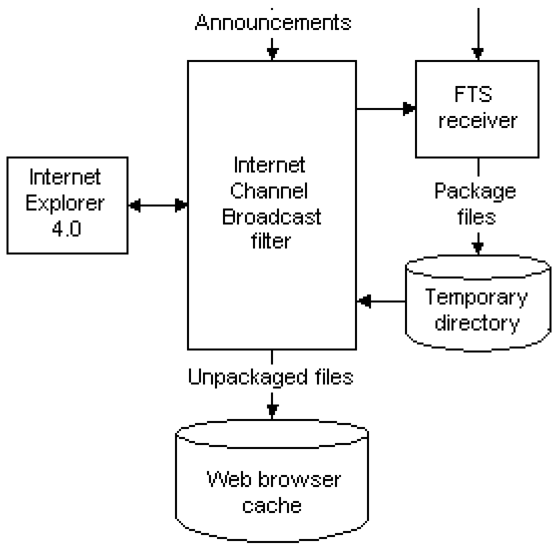
After the FTS receiver has stored the packaged files in the receiving directory, the client filter unpackages and reconstructs the files and stores them in the Internet Explorer cache. If a package transfer is unsuccessful, the incomplete package file is deleted. The files that the filter stores in the cache are marked as Subscription Content so they will not be inadvertantly deleted if the cache becomes full.

**Note** When the filter unpackages the received files, it prevents unpackaged files from specifying a destination directory above the receive directory. In other words, an incoming file cannot specify a destination directory such as ..\..\..\Windows\System. Files will only be unpackaged to the receive directory and its subdirectories.

Once the subscription update is received, the filter indicates that the channel has been updated by adding a gleam, or red dot, to the channel icons in the channel bar and in the Favorites/Channels menu. The filter also checks whether the subscription update schedule is set to manual. If it is not, the filter changes the setting to manual. This prevents Internet Explorer from collecting the subscription content from the Web. Since the subscription is being updated by Internet Channel Broadcasting, it would be redundant for Internet Explorer to also update the subscription files.

The following illustration shows the function of the Internet Channel Broadcast filter in the client architecture.





Under typical operation, the Internet Channel Broadcast filter is automatically started when the Broadcast Architecture client software is installed. The filter usually does not require any further maintenance after installation. However, if you need to stop or start the filter, or to check when the filter last matched an announcement to a file group, you can use the Announcement Filter Manager. You can also pause the filter by right-clicking the Announcement Listener icon on the taskbar and clicking **Pause Announcement Listener**. Note that doing so pauses all announcement filters running on the broadcast client.

Transfer events are logged in the log directory, typically C:\Program Files\Webcast\Logs. By default, this directory is not created during installation. If you want the Internet Channel Broadcast client to generate log files, you must create the log directory manually. You can change the location of the log directory by changing the value of the **log_dir** registry key. For more information, see Internet Channel Broadcast Client Registry Keys and Testing Data Reception.

## Using the Internet Channel Broadcast Client

[This is preliminary documentation and subject to change.]

The following topics describe how to use and programmatically extend the Internet Channel Broadcast client.

- Configuring a Service Provider
- Subscribing to a Channel
- Editing a Subscription
- Deleting a Subscription
- Creating a Custom Client Application
- Detecting an Internet Channel Broadcast Client
- Testing Data Reception

## Configuring a Service Provider

[This is preliminary documentation and subject to change.]

An Internet Channel Broadcast service provider is a person or organization that administers an Internet Channel Broadcast server, which transmits channel announcements and updates. The Internet Channel Broadcast client supports multiple service providers.

Before the client can receive announcements from a service provider, the provider's announcement address must be configured in the client computer's registry. This configuration is done by adding a string value to this registry key:

**HKLM\Software\Microsoft\TV Services\Announcements**

This value must be formatted as *xxx***.***xxx***.***xxx***.***xxx***:***yyyy* where *xxx***.***xxx***.***xxx***.***xxx* specifies the Internet Protocol (IP) address over which the service provider transmits announcements, and *yyyy* is the corresponding port.

A service provider should provide a mechanism to add this announcement configuration string to a user's registry when the user signs up for the service. If such a mechanism is not provided, the user must add the value to the registry manually.

## Subscribing to a Channel

[This is preliminary documentation and subject to change.]

The Internet Channel Broadcast client uses the channel subscription model provided by Microsoft® Internet Explorer version 4.0. This model means that you can create subscriptions for Internet channel broadcasting in exactly the same manner that you create Internet Explorer subscriptions.

**To subscribe to a channel**

1. In Internet Explorer, click the **Channels** button on the toolbar.
2. In the Explorer bar, click **Channel Guide**, and then follow the instructions on your screen.

For more information about creating subscriptions with Internet Explorer 4.0, see Further Information on Data Services for the Client.

Future releases of Broadcast Architecture may provide a means to configure a service provider at the same time that the user subscribes to a channel. In this scenario, when a user with Broadcast Architecture installed subscribes to a Web site, future versions may present a list of service providers.

These service providers may include, for example, local television stations broadcasting Internet channels over the *vertical blanking interval* (VBI). The user might then select from the list a provider from which to receive updated channel information. At that point, the service provider settings might be configured automatically. For more information, see Configuring a Service Provider.

## Editing a Subscription

[This is preliminary documentation and subject to change.]

The Internet Channel Broadcast client uses the Internet Explorer 4.0 channel subscription model. This model means that you can edit Internet channel broadcast subscriptions in exactly the same manner that you edit Internet Explorer subscriptions.

**To edit a subscription**

1. In Internet Explorer, click **Manage Subscriptions** on the **Favorites** menu.
2. Right-click the subscription you want to update, and then click **Properties**.
3. Specify the settings you want for receiving and scheduling the subscription.

For more information about editing subscriptions with Internet Explorer 4.0, see Further Information on Data Services for the Client.

## Deleting a Subscription

[This is preliminary documentation and subject to change.]

The Internet Channel Broadcast client uses the Internet Explorer 4.0 channel subscription model. This model means that you can delete Internet channel broadcast subscriptions in exactly the same manner that you delete Internet Explorer subscriptions.

**To delete a subscription**

1. In Internet Explorer, click **Manage Subscriptions** on the **Favorites** menu.
2. Right-click the subscription you want to update, and then click **Delete**.

For more information about deleting subscriptions with Internet Explorer 4.0, see Further Information on Data Services for the Client.

## Creating a Custom Client Application

[This is preliminary documentation and subject to change.]

If the functionality provided by the Internet Channel Broadcast client does not suit your needs, you can create a custom client application or custom subscription application.

A client application must include an announcement filter able to receive Internet channel broadcast announcements. The client application must be able to discern whether the user is subscribed to a particular channel, and to receive and unpackage *File Transfer Service* (FTS) transmissions from the Internet Channel Broadcast server.

A subscription application should display a list of channels available for subscription, enable users to subscribe to channels, maintain an inventory of subscribed channels, and provide a mechanism by which users can delete or edit their subscriptions.

## Detecting an Internet Channel Broadcast Client

[This is preliminary documentation and subject to change.]

Internet content providers can parse the HTTP headers transmitted by incoming browsers to test whether the browser is running on a computer that has the Broadcast Data Services component of the Microsoft® Windows® 98 operating system installed. This enables Web servers to return different content depending on whether the user can receive broadcast data.

For example, a Web server that provides Internet subscription channels could redirect a broadcast-enabled user to a channel definition file (CDF) that contains more and bigger files, knowing that the user will be able to use high-bandwidth broadcast delivery instead of downloading over the Internet.

To detect whether a Web browser is running on an broadcast-enabled computer, check whether the HTTP_ACCEPT header string contains "application/MS-BPC". If this string is present, the computer is a broadcast client.

The following example uses Active Server Pages to test whether a user has the Broadcast Data Services component of Windows 98 installed. If they do, the server displays a link to a broadcast-enriched Internet channel. Otherwise, the server displays a link to a default channel.

```
<HTML>
<HEAD><TITLE>ICS Client Test Page</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF">
<%
  Dim acptstr, resstr

  'read in the HTTP_ACCEPT header string
  acptstr = Request.ServerVariables("HTTP_ACCEPT")
  'check whether "MS-BPC" is included in the header string
  'if "MS-BPC" is present, the client is broadcast-enabled
  resstr = InStr(1, acptstr, "MS-BPC")

  If resstr = 0 Then
    'Insert non-broadcast enabled code
    Response.Write ("You are not broadcast-enabled.")
    Response.Write ("Click <A HREF="NoBroadcast.cdf">here</A> to subscribe.")
  Else
    'Insert broadcast-enabled code
    Response.Write ("You are broadcast-enabled.")
    Response.Write ("Click <A HREF="Broadcast.cdf">here</A> to subscribe.")
  End If
%>
</BODY>
</HTML>
```

For more information about Active Server Pages, see Further Information on Data Services for the Client.

## Testing Data Reception

[This is preliminary documentation and subject to change.]

The following section is intended for users who wish to test whether their Internet Channel Broadcast Client is correctly receiving subscription data. The techniques described could be used by content providers checking their transmission process, or users who want to verify that their installation and configuration of the client.

In order to ensure that the subscription content that arrives on your client computer is being received from an Internet Channel Broadcast Server instead of by some other means, such as Internet Explorer webcasting, you should set the subscription to be updated manually. This prevents Internet Explorer from updating the files in your cache, and still allows the subscription files to be updated by the broadcast client.

**To set a subscription to manual update**

1. In Windows Explorer, open the C:\Windows\Subscriptions\ directory.
2. Right-click the subscription you want to set to manual update and select **Properties** from the menu that appears.
3. Click the **Schedule** tab.
4. Click the **Manually** radio button, and click **OK**. The subscription is only automatically updated by the Internet Channel Broadcast client.

You can easily tell when the client updates a subscription. The client will mark the icon of updated subscription with a gleam, or red dot. The gleam appears on both the the Channel-bar icon for the subscribed item as well as the icon displayed on the Favorites/Channel menu for the subscribed item.

If you require more details about the client's data reception, you can enable the logging feature of the Internet Channel Broadcast client. This feature causes the client to write status information to a text file each time it receives incoming data.

**To enable client logging**

1. Create a directory in the location specified by the HKLM/Software/Microsoft/Webcast/General**log_dir** registry value. Typically, this is "C:\Program Files\Webcast\Logs".
2. Using Regedit.exe, add a new DWORD value named **loglev** under HKLM/Software/Microsoft/Webcast/General. Set this value to the log level that corresponds to the level of verbosity you want. A setting of 1 indicates normal logging, whereas a setting of 3 indicates a higher level of detail. Note that setting the logging level to a value such as 5 can quickly fill up your computer's storage.
3. Restart your computer.

   The client will now write log data to the newly-created log directory.

**To view logged data**

- Open the file in the log directory using a text editor.

# Internet Channel Broadcast Client Reference

[This is preliminary documentation and subject to change.]

The following sections provide reference information for the Internet Channel Broadcast client:

- Internet Channel Broadcast Client Registry Keys
- Internet Channel Broadcast Announcement Format

## Internet Channel Broadcast Client Registry Keys

[This is preliminary documentation and subject to change.]

The Internet Channel Broadcast client reads the registry to initialize its settings. The following keys are added to the registry when the Broadcast Data Services component of the Microsoft® Windows® 98 operating system is installed. Unless a user or application changes these values, these keys contain the default values specified in the following tables.

Internet channel broadcasting entries are found in this key:

**HKLM\Software\Microsoft\Webcast\**

This key contains two subkeys with values that affect the Internet Channel Broadcast client, **\General** and **\client**.

The following client values are found in the **\General** subkey.

| **Value** | **Type** | **Meaning** |
|---|---|---|
| **bin_dir** | String | The directory that contains the Internet Channel Broadcast filter and other executable files used by the Internet Channel Broadcast client. The default is C:\Program Files\Webcast\Bin. |
| **log_dir** | String | The directory to which the Internet Channel Broadcast client writes log files. The default is C:\Program Files\Webcast\Logs. By default, the log directory is not created on the user's computer when the client is installed. If you want the client to log status, you must manually create a log directory. |
| **webcast_dir** | String | The directory where the Internet Channel Broadcast client is installed. If this directory is not set, the default is C:\Program Files\Webcast. |

The subkey **\client** contains the following values.

| **Value** | **Type** | **Meaning** |
|---|---|---|
| **recv_dir** | String | The directory where received files are put until they are unpackaged. The default is C:\Program Files\Webcast\Recv. |
| **loglev** | **DWORD** | A value that sets the level of verbosity for information sent to the log file. Zero specifies no logging; 1 (one) is usual verbosity; values up to 6 are increasingly verbose.<br><br>Note that setting the log level to a high value can quickly fill up your hard drive. For example, a **loglev** setting of 5 can easily generate a 10 megabyte file overnight. |

**Note** When you open a file, Microsoft® Internet Explorer attempts to determine whether the files in its cache need to be refreshed from the Internet. In other words, when you open a cached Web file, Internet Explorer connects to the Internet. If an updated version of the file is available, Internet Explorer automatically downloads it. This functionality can make it difficult to test the Internet Channel Broadcast client. To prevent Internet Explorer from automatically updating cached files when you open them, click **Work Offline** on the **File** menu in Internet Explorer.

## Internet Channel Broadcast Announcement Format

[This is preliminary documentation and subject to change.]

Internet channel broadcast announcements are transmitted to the client by the Store-and-Forward server operated by the content provider. These announcements inform the client of upcoming data transmissions.

The announcements are formatted using Session Description Protocol (SDP) protocol. For more information, see Announcement Format. This topic discusses only the SDP fields used by Internet channel broadcast announcements.

The following is a sample Internet channel broadcast announcement:

```
v=0
o=Webcast3 3084034855 3084023432 IN IP4 157.56.137.117
s=Webcast package, group ChannelName
i=Webcast package, group ChannelName, URL range http://FirstFile.htm to http://Last
u=http://Channel.cdf
c=IN IP4 233.17.43.2/1
t=3084034855 3084035155
a=webcast_version:4.10.1592
a=length: 1067
a=urlroot: http://Channel.cdf
a=group: ChannelName
a=package-type: CDF_CHANNEL
a=cdf-update: TRUE
a=first-url: http://FirstFile.htm
a=last-url: http://LastFile.htm
a=page-count: 1
a=pkg-version: 3
m=webcast 1781 FTS 0
```

# Broadcast Server Architecture

[This is preliminary documentation and subject to change.]

The following sections provide an overview of the type of server architecture used to transmit broadcast data over a broadband network to *broadcast clients* in the home. The *head end* in this discussion is the physical infrastructure that gathers, coordinates, and broadcasts data over such a network.

The head-end server architecture relies on the following three software components to deliver broadcast data to transport mediums such as cable systems, satellite uplinks, or terrestrial antennas:

- Content server application. This application, written by a content provider and running on a computer at the head end, gathers, schedules, and sends data to the Microsoft Multicast Router (MMR), as described in Writing Content Server Applications.
- Microsoft Multicast Router (MMR). This program routes data from the content server application to the output driver. The MMR can forward Internet Protocol (IP) multicast packets directly to the output driver or it can forward packets embedded in a Transmission Control Protocol (TCP) stream from the content server application.
- Output system software. This software is a dynamic-link library (DLL) file that the MMR calls to send data to the transport medium. Each transport medium requires a unique output driver DLL. A single MMR program can use more that one output driver DLL to enable it to send data on multiple transport media.

# Content Server Application

[This is preliminary documentation and subject to change.]

A content server application generates a data stream that is sent to the broadcast client. An example of a content server application is the Internet Channel Broadcast Server, which sends World Wide Web pages. This application collects Hypertext Markup Language (HTML) pages from the Web, using a mechanism based on Internet Explorer version 4.0 channels. Then, the server encapsulates the pages in Transmission Control Protocol/Internet Protocol (TCP/IP) packets and passes them to the MMR.

There are many factors to consider when developing a content server application. Such factors include the amount of data to send, how often the data should be resent, and the security the data requires. For information on how these and other factors affect the design of a content server application, see Writing Content Server Applications.

# Microsoft Multicast Router

[This is preliminary documentation and subject to change.]

The Microsoft Multicast Router (MMR) receives data from the content server application and routes it to the output driver. The MMR program is divided into two parts that are covered in the following sections.

- The Router Service Program is a server process that handles network packets.
- The Router Manager is a program that you use to control the router service program.

For more information about using the MMR, see Working with the Microsoft Multicast Router.

## The Router Service Program

[This is preliminary documentation and subject to change.]

The router service program (Mrouter.exe) is the program that listens for packets coming from the content server application and forwards broadcast data to the output driver.

There are two possible ways for a content server application to send data to the MMR. The first is by sending IP multicast packets using an IP address on which the MMR expects data. The MMR passes these packets directly to the output driver for transmission. The content server is responsible for sending data at times and speeds that are acceptable to the transmission medium. For example, if the output driver is designed to send data to a *vertical blanking interval* (VBI) data inserter, the content server application must send the data at a rate slow enough for the VBI data inserter to keep up. In addition, if there is another content server application using the same output driver, both content server applications must be designed so that they do not send data at the same time.

The second way for a content server to send data to the MMR is through a TCP/IP tunnel. A TCP/IP tunnel is a TCP/IP connection between two applications that is used to carry other network data. In the case of the MMR, this data is UDP packets destined for the output driver. The advantage of using a TCP/IP tunnel is that it gives the MMR an opportunity to send messages back to the content server application. These messages tell the content server application when it is sending too much data or sending data at the wrong time.

## The Router Manager

[This is preliminary documentation and subject to change.]

The router manager (Mmradm.exe) is a program that lets you monitor the status and set parameters for the router service program. Mmradm.exe does not have to run on the same computer as the router service program. The only requirement is that a network connection exists between Mmradm.exe and the system running Mrouter.exe.

## Output System Software

[This is preliminary documentation and subject to change.]

The output driver is a dynamic-link library (DLL) file that contains routines that the MMR can call to send packets to the broadcast hardware. The MMR keeps a list of which packets go to which output driver. For example, you can use Mmradm.exe to setup the MMR to send all packets arriving through a certain tunnel to the VBI output driver. The output driver copies the packets to the VBI device and signals the MMR when it can accept more data. To read about how to write an output driver module, see Sample Output Driver DLL.

# Writing Content Server Applications

[This is preliminary documentation and subject to change.]

Before establishing a *data service* using Broadcast Architecture, a *content provider* must generally create or acquire computer software to manage the delivery of the data from its source to its audience. This topic discusses some of the design and programming issues that a development team needs to address when creating a data service system.

The best programming strategy depends to some extent on what kind of data is involved, as is discussed in two sections reviewing the possible kinds of broadcast data:

- Broadcast Data Categories
- Broadcast Data Characteristics

The section Main Office Software describes the software that content providers must write or aquire to process and broacast data.

Also, where appropriate, a content provider must establish an online server for managing viewer subscriptions, purchases, and responses. Although the needs of such a server should be taken into account when planning development, the specific functions of any given server are beyond the scope of this document.

## Broadcast Data Categories

[This is preliminary documentation and subject to change.]

Data intended for the Broadcast Architecture falls into three general categories:

- Internet channel broadcasting
- Enhanced television
- Subscription data

The first two categories are handled using special-purpose tools supplied with Microsoft® Site Server. The Internet Channel Broadcast Server handles the first catagory, and most enhanced television broadcasts will initially be created in existing television studios and transmitted as analog *NTSC* streams carrying the enhancement data in the *vertical blanking interval* (VBI). For more information on Internet channel broadcasting, see Internet Channel Broadcast Client and Internet Channel Broadcast Server. For more information on VBI enhancements, see Serial VBI.

The third category, subscription data, here means any content other than World Wide Web content and television enhancements that the user of a broadcast client chooses to receive. The choice to subscribe need not carry a price, and the choice may not even necessarily be explicit. For example, tuning to a given channel at a given time might be interpreted as expressing interest in receiving related data.

Subscription data can take many different forms, including the following:

- Digital video for sale or rental
- Computer software, including games
- High-resolution still images or animations
- Digital magazines or newspapers
- Digital mail-order catalogs or advertisements

Because subscription data can vary so much in format, audience, and purpose, custom software is required to manage its broadcast.

## Broadcast Data Characteristics

[This is preliminary documentation and subject to change.]

Data suitable for broadcast falls into many different categories and can have a variety of formats. The following general characteristics of different data have important implications for the software needed to generate, transmit, and receive the corresponding broadcast streams:

- *Value*. How much is the data worth? A very high value probably justifies better encryption, more error correction, and more frequent transmission. Low value may allow higher compression and less frequent transmission.
- *Precision*. How much of the data's value depends on its complete and accurate transmission? When a software program is transmitted, for example, every bit must be correctly received and verified, or the data is useless. When a bitmap graphic is transmitted, on the other hand, a good deal of data can usually be lost without compromising the value of the image. Note that because the client cannot notify the host of lost data in Broadcast Architecture, incomplete and damaged data is always discarded. Partial data and data containing irrecoverable errors is simply not received.
- *Transience*. Some data loses its value soon after it is transmitted, for example stock prices or enhancements to a television show. Other data loses value very slowly, for example a software program or reference work.
- *Urgency*. How important is it that viewers receive the data within a given time-frame? Urgency can be understood as a function of value and transience, but it also has a psychological component deriving from viewers' perceptions. For example, a user purchasing a new software program expects it to come shortly after she pays for it. If, on the other hand, the user purchases regular updates to her encyclopedia articles, she does not expect an immediate response.
- *Expiration*. How long may a viewer keep the data? Transient data is usually used immediately upon reception and discarded, but other data may be kept indefinitely. The viewer's rights to store and use data are often limited by licensing agreements, as in the case of software rentals, pay-per-view programs, or limited subscriptions.

- *Segmentation*. How is the data divided up, particularly from a subscription standpoint? A viewer might sign up for a persistent stream of information, as in the case of a stock-price or news service; for individual files, as in the case of downloading software; or perhaps for filtered subsets of a complex data stream.
- *Homogeneity*. How homogeneous is the data? Stock price data, at one extreme, usually consists of small, uniform data objects that all have exactly the same size and format. A news stream, at the other extreme, might consist of many different types of data objects of radically different sizes and formats.

Analyzing a data service in terms of these characteristics is useful in determining what software is needed in the content provider's studio, at the broadcast head end, and on the home computer in order to manage the data most effectively.

## Main Office Software

[This is preliminary documentation and subject to change.]

The software tools for content development typically run on computers located in the content provider's own facilities and vary widely with the type, source, and design of the content itself. Some of these tools are off-the-shelf commercial products, while others are special-purpose applications.

Some of the tasks content providers may need to perform follow:

- Creating Content Data
- Assembling Content Data
- Storing and Cataloging Content Data
- Synchronizing Enhancements with Television
- Scheduling Broadcasts
- Specifying Announcements
- Forwarding Content data to the Microsoft Multicast Router

### Creating Content Data

[This is preliminary documentation and subject to change.]

Much content data used to enhance television will probably be created from scratch, using existing World Wide Web design and programming tools. To locate more information on creating content data, see Further General Information.

## Assembling Content Data

[This is preliminary documentation and subject to change.]

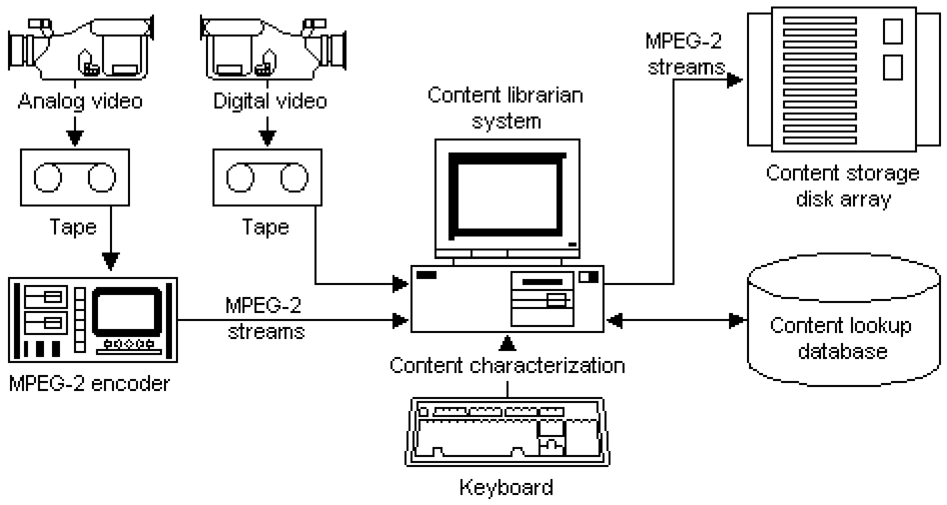
This step may involve collecting data from the World Wide Web, from a data warehouses, or files on a corporate network. It is important to develop an efficient strategy for collecting data to be broadcast, whatever its source.

## Storing and Cataloging Content Data

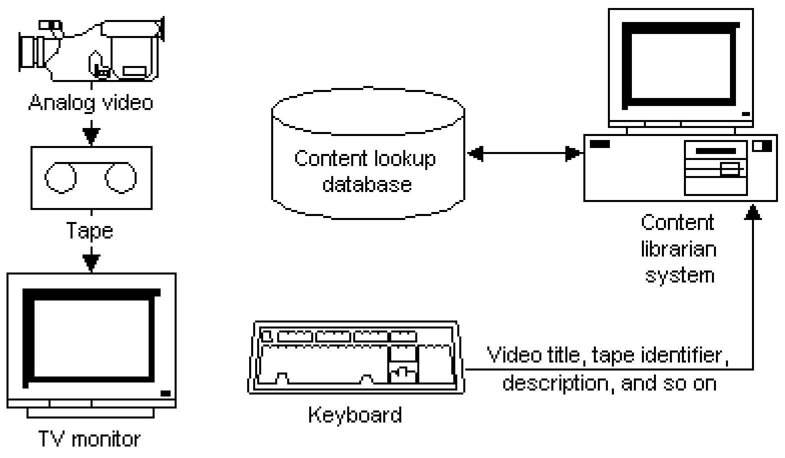
[This is preliminary documentation and subject to change.]

The first step in processing broadcast data is to catalog it. This is true whether the data is analog television, digital video, or any other form of digital data. This task can be accomplished using a database that contains information about what data is available and where to find it.

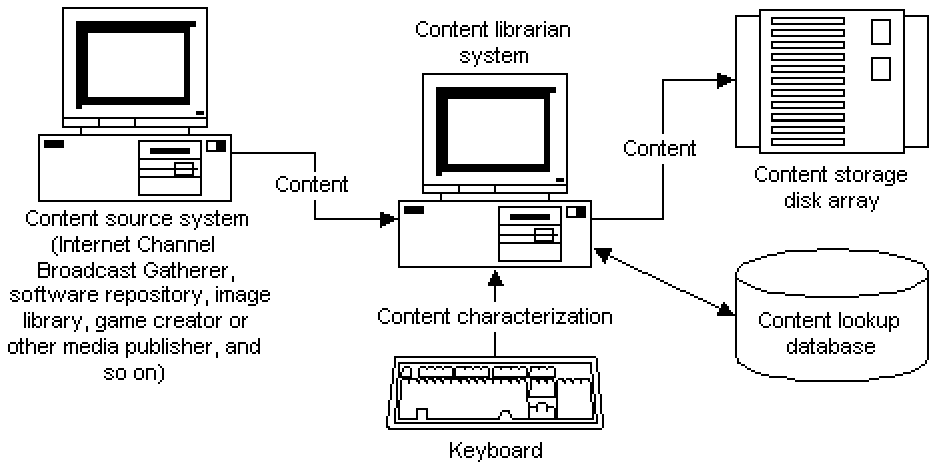
The diagrams following provide a schematic representation of two possible systems that catalogs content. The first diagram, following, illustrates cataloging and storing digital television or audio content. The content librarian system is a computer that an operator uses to collect the digital content, characterize it with key words and descriptive text, and store the content data and related information.



The following diagram illustrates cataloging and storing analog television content. In this case the content data is not directly available to the content librarian system, so the content lookup database must include identifying information. This information could include the title of the video, a tape ID, or a description.



The following diagram illustrates cataloging and storing other digital content. This system is similar to the digital audio and video system. The difference is that the content data comes from other sources source as web page gatherers or image libraries.


Content source system (Internet Channel Broadcast Gatherer, software repository, image library, game creator or other media publisher, and so on)
Content
Content librarian system
Content
Content storage disk array
Content lookup database
Content characterization
Keyboard

## Synchronizing Enhancements with Television

[This is preliminary documentation and subject to change.]

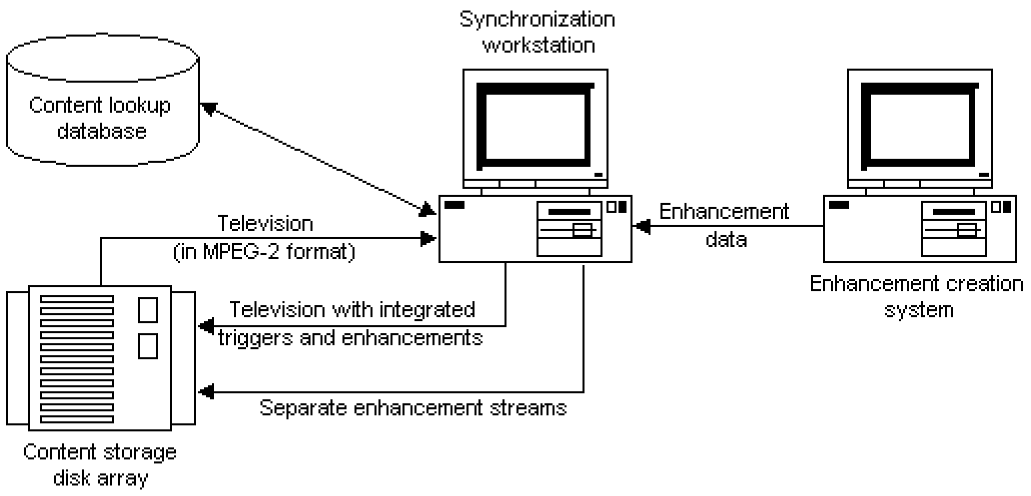
The step of synchronizing television enhancements to shows is primarily relevant for digital enhancements to television and audio programming.

One way to synchronize digital broadcast data with television is to embed hidden triggers in the broadcast data stream. A program running on the broadcast client can check for triggers and display the appropriate digital enhancements when they arrive. To take advantage of this technique, a content provider must insert appropriate triggers in the broadcast data stream and settle on a way to associate each trigger with corresponding enhancement data.

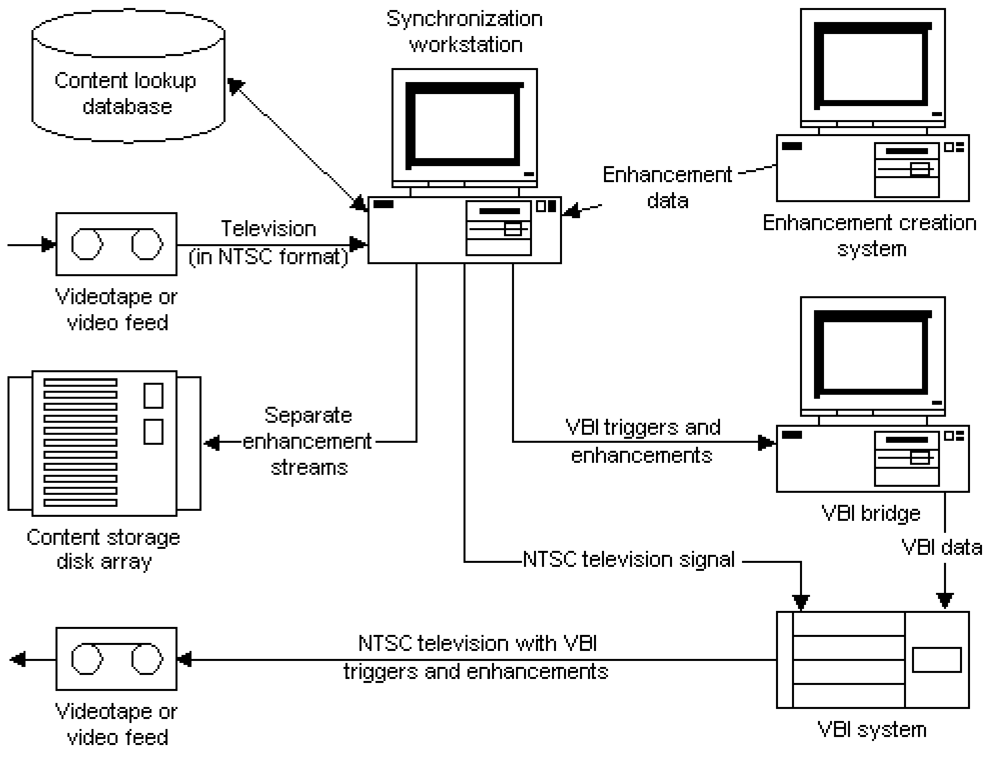
In early implementations of Broadcast Architecture, digital enhancements for television will usually be stored in the *vertical blanking interval* (VBI) of analog television signals. However, in later implementations of Broadcast Architecture, there may be alternate, richer data steams broadcast in parallel with the television show across digital channels.

Furthermore, even early implementations of Broadcast Architecture may broadcast in advance of a show one or more digital files of subscription data, so these files are available when the show airs. This last possibility is particularly attractive to advertisers because it allows more graphics and information to be transmitted than would fit in the VBI bandwidth available..

Tools to synchronize such ancillary enhancements with television should update the content lookup database so that the link between the television show and the enhancements is recorded in that database. This link can be used to make sure that the enhancements for a particular show get transmitted in time to be used during the show. The following diagrams show such a synchronization process. The first illustration shows how digital data is combined with enhancements. First, an enhancement creation system sends the enhancements to the synchronization workstation. This workstation uses information in the content lookup database to combine the television data from the content storage array with the enhancement data. Then the synchronization workstation returns the combined data and any separate enhancement streams, to the content storage array.


Synchronization workstation
Content lookup database
Television (in MPEG-2 format)
Enhancement data
Enhancement creation system
Television with integrated triggers and enhancements
Separate enhancement streams
Content storage disk array

In following illustration enhancements are synchronized with analog television. The enhancement creation system, synchronization workstation, and content lookup database function in the same way as the previous illustration. The difference is in the source of the video data and how the final output is stored. With analog video, only the separate enhancement streams are stored in the content storage disk array. The synchronization workstation gets video from a video tape or other video feed and passes it to a VBI system. At the same time, the synchronization workstation passes the VBI triggers and enhancements to a VBI bridge that, in turn, passes the data to the VBI system. The VBI system combines the video with the VBI data and outputs the NTSC television with triggers and enhancements.


Synchronization workstation
Content lookup database
Enhancement data
Enhancement creation system
Television (in NTSC format)
Videotape or video feed
Separate enhancement streams
VBI triggers and enhancements
Content storage disk array
VBI bridge
VBI data
NTSC television signal
NTSC television with VBI triggers and enhancements
Videotape or video feed
VBI system

## Scheduling Broadcasts

[This is preliminary documentation and subject to change.]

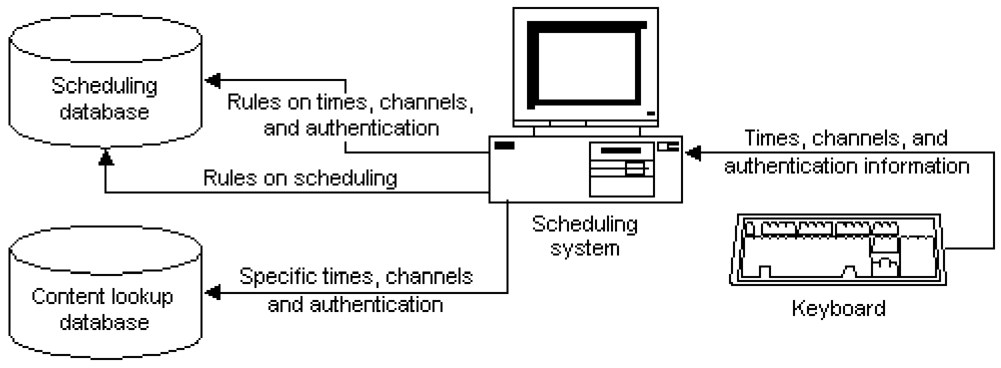
In some cases, scheduling when broadcast data is transmitted can be trivial. In the case of a stock ticker service, for example, the data is transmitted as soon as it arrives. In many cases, however, scheduling is an extremely complex task, particularly if bandwidth is limited. Trade-offs may need to be made balancing value, urgency, precision, and data size. For example, if a data file is large, you may have to send it less often, sacrificing speedy delivery and the amount of error correction data that can be sent. Tools to facilitate this process will be extremely useful. Some considerations regarding creating such tools follow.

When the decision is made to broadcast a specific content file, relevant information must be placed in the content lookup database. This information includes the data service over which the broadcast is to occur, any authentication information required to secure bandwidth on that data service, the time or times of broadcast, the amount of bandwidth required for the broadcast, and any other information needed to ensure that the broadcast goes out over the right channel at the right time.

A scheduling system should also take into account any dependencies recorded in the content lookup database, allowing an operator to schedule these dependencies at the same time. For example, if a television show required some enhancements to be preloaded, the scheduling system should transmit such enhancements before the show.

Because scheduling is a common and repetitive task, as much as possible of the relevant scheduling information should be stored in a scheduling database. Applications running on the head end can retrieve this information automatically to support scheduling of individual content files.

The following diagram illustrates the components of this kind of scheduling process.


Scheduling database
Rules on times, channels, and authentication
Rules on scheduling
Times, channels, and authentication information
Scheduling system
Keyboard
Content lookup database
Specific times, channels and authentication

## Specifying Announcements

[This is preliminary documentation and subject to change.]

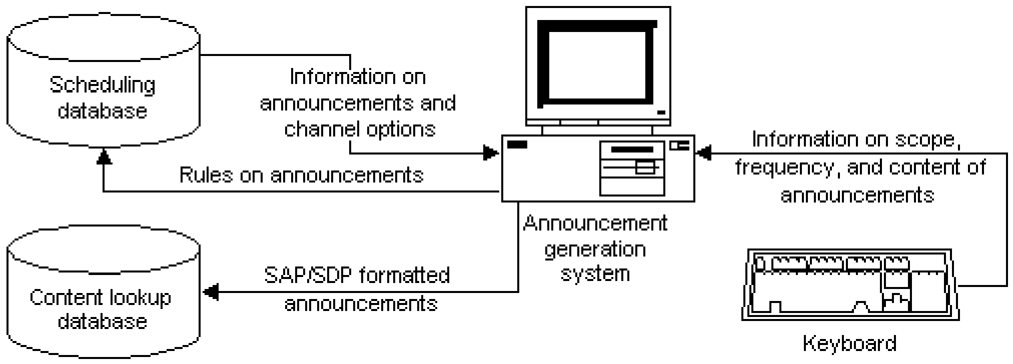
Not only does a broadcast need to be scheduled, but its announcement strategy needs to be determined, and its announcement text specified.

Typically, two types of announcement are available for every broadcast: a *global announcement* and a *local announcement*. A global announcement is usually issued infrequently over a broad range of tuning space, so as to alert client machines that a given content file will be broadcast over a given channel at a given time.

A local announcement, on the other hand, is usually sent frequently for a short period before a broadcast occurs, over the same channel on which the broadcast will occur. Its purpose is to alert a client that the broadcast is imminent and to provide any last-minute tuning information that may be needed.

Both types of announcement must be formatted as *Session Announcement Protocol* (SAP)/*Session Description Protocol* (SDP) pairs

The following diagram illustrates an announcement generation system. For more information on the use of a scheduling database, see Scheduling Broadcasts; for more information on the use of a content lookup database, see Storing and Cataloging Content Data.


Scheduling database
Information on announcements and channel options
Rules on announcements
Information on scope, frequency, and content of announcements
Announcement generation system
SAP/SDP formatted announcements
Content lookup database
Keyboard

An announcement generation system should enable an operator to choose the timing and frequency of both global and local announcements, to specify associated Internet Protocol (IP) addresses for each type of announcement, and to determine the content of each type of announcement. This information can then be stored in the content lookup database in SAP/SDP format, ready to use.

For more information on how Broadcast Architecture works with announcements, see Announcement Listener.

## Forwarding Content Data to the Microsoft Multicast Router

[This is preliminary documentation and subject to change.]

A content server program forwards a file to be broadcast to the Microsoft Multicast Router at times, and on IP addresses indicated in the content lookup database.

The information used to determine what time and IP address to use might include:

- The data service and channel on which the stream of data from the file should be broadcast.
- Bandwidth assurance information: Should bandwidth be reserved, or can this stream be broadcast *opportunistically*?
- Bandwidth requirements: If bandwidth needs to be reserved, how much and for how long?
- The specific IP multicast address to be reserved, if one is necessary.
- The time or times this stream should be broadcast.
- The global announcement associated with this stream, if any, along with a specification of when, how many times, and at what intervals the announcement should be sent.
- The local announcement associated with this stream, if any, along with a specification of when, how many times, and at what intervals the announcement should be sent.
- Stream reuse information: Should this stream be saved for future broadcast or deleted after sending?

The content server application is responsible for such things as error checking, and error correction, encryption, key generation, and key encryption, when this is appropriate.

The content service provider communicates with the Microsoft Multicast Router (MMR) through

TCP/IP. Using TCP/IP lets the content server application run on any computer that has a TCP/IP network connection to the MMR computer.

# Working with the Microsoft Multicast Router

[This is preliminary documentation and subject to change.]

The Microsoft Multicast Router (MMR) is a network server application that provides services to forward network traffic from a local computer network to a variety of unidirectional broadcast encoders.

For more information about the MMR, including how to install and run it, see the following topics:

- About the Microsoft Multicast Router
- Installing the Microsoft Multicast Router
- Configuring the Microsoft Multicast Router

## About the Microsoft Multicast Router

[This is preliminary documentation and subject to change.]

The Microsoft Multicast Router (MMR) enables content servers to send data over your broadcast head end. The MMR is made up of a router program (Mrouter.exe), an operator interface (Mmradm.exe), and an output driver. The router program receives packets to be transmitted and passes them on to the output driver according to the operator instructions. Each type of transmission uses a specific output driver. For example, to send data in the *vertical blanking interval* of a television signal, the MMR uses Vbi_out.dll.

**Note** Some beta versions of the Microsoft Multicast Router call output drivers virtual interfaces (VIFs) or output subsystems.

## Installing the Microsoft Multicast Router

[This is preliminary documentation and subject to change.]

The requirements for the Microsoft Multicast Router (MMR) are:

- A computer based on a Pentium 150-megahertz, or faster, CPU.
- At least 32 megabytes of RAM are required. 64 megabytes of RAM are recommended
- Microsoft® Windows NT® Server version 4.0, or newer, operating system with the remote procedure call (RPC) service installed

- A network card connected to a local source network
- Appropriate hardware to interface with the broadcast head end

▶ **To install the MMR software**

To install Mrouter.exe, run the setup program to copy the MMR files to the server computer, then enter this command from the install directory:

**mrouter -install**

If you configure the MMR to forward multicast packets, you must install the MSDBN NDIS Packet Driver in Windows NT. Follow these steps to install the driver:

1. In the Windows NT Control Panel, double-click **Network**.
2. Click the **Protocols** tab.
3. Click **Add** button to bring up the **Select Network Protocol** dialog box.
4. Click the **Have Disk** button.
5. Type the path to the broadcast server files, and click **OK**.
6. Select **MSBDN NDIS Packet Driver** from the list of components.
7. Follow the prompts to install the component.
8. Restart the computer.

# Configuring the Microsoft Multicast Router

[This is preliminary documentation and subject to change.]

The Microsoft Multicast Router (MMR) uses the program Mmradm.exe to monitor the status of the MMR and to set configuration parameters. You can use Mmradm.exe to configure any running copies of Mrouter.exe on a local area network from any computer on the network.

The first step in configuring the MMR is to add the MMR to the list of routers in the main window of Mmradm. If Mmradm is running on the same computer as the MMR, the router automatically appears in the window. Otherwise you can select add from the file menu to add a router.

The next step is to select the router and use the configure command from the file menu. There are several tabs in the configuration dialog box that let you set various parameters for the router. Each router must have at least one output driver before it can send data. Use the **Output Subsystem** tab to add the output driver to the router.

# MSBDN Reference

[This is preliminary documentation and subject to change.]

This section documents Microsoft Broadcast Data Network (MSBDN) functions. A *content server application* running at a broadcast system head end would use these functions to transmit data across a Microsoft Multicast Router (MMR) to the broadcast network.

These functions include:

- Tunnel functions
- Data transfer functions

For an overview of the head-end server architecture within which the MSBDN functions operate, see Broadcast Server Architecture.

For more information about how to design a content server application, see Writing Content Server Applications.

## Tunnel Functions

[This is preliminary documentation and subject to change.]

Tunnels are Windows Socket connects that contain other data channels. In the case of content server applications, the content server application can open a tunnel to the MMR that contains the IP multicast packets to transmit. The purpose for using a tunnel instead of directly multicasting is to provide a back channel from the MMR to the content server application. The MMR can use this back channel to control the rate at which the content server application send data.

| **Function** | **Description** |
|---|---|
| msbdnOpenTunnel | Opens a tunnel to the MMR |
| msbdnClose | Closes a tunnel |
| msbdnOpenTunnelEx | Opens a tunnel to the MMR using a specific reservation |

## msbdnOpenTunnel

[This is preliminary documentation and subject to change.]

The **msbdnOpenTunnel** function opens a tunnel to the MMR.

```
EXTERN_C MSBDNAPI HBDNCONN msbdnOpenTunnel(
  LPCSADDR_INFO        lpcsMMRAddr,
  LPSOCKET_ADDRESS    lpDestAddr,
  WORD              wTTL,
  DWORD              dwReservationID
);
```

**Parameters**

*lpcsMMrAddr*
The name of the MMR to connect to.
*lpDestAddr*
The socket address to use for the tunnel.
*wTTL*
Time to live value to use with the socket interface.
*dwReservation ID*
Identify the bandwidth reservation to use.

**Return Values**

Returns a handle to the tunnel connection or NULL if the connection cannot be made. Use **GetLastError**() to get more information when **msbdnOpenTunnel** returns NULL.

**See Also**

msbdnClose, msbdnSend

# msbdnClose

[This is preliminary documentation and subject to change.]

The **msbdnClose** function closes a connection to the MMR

```
MSBDNAPI BOOL WINAPI msbdnClose(
  HBDNCONN hBdnCon
);
```

**Parameters**

*hBdnCon*
Handle to the connection to close.

**Return Values**

If an error occurs while closing the connection, **msbdnClose** returns SOCKET_ERROR.

### See Also

msbdnOpenTunnel, msbdnOpenTunnelEx

# msbdnOpenTunnelEx

[This is preliminary documentation and subject to change.]

The **msbdnOpenTunnelEx** function opens a tunnel to the MMR using a specific reservation.

```
MSBDNAPI HBDNCONN WINAPI msbdnOpenTunnelEx(
  LPCSTR lpcsMMrAddr,
  LPCSTR lpDestAddr,
  WORD wDestPort,
  WORD wTTL,
  DWORD dwReservationID
);
```

### Parameters

*lpcsMMrAddr*
: The name of the MMR to connect to.

*lpDestAddr*
: The socket address to use for the tunnel.

*wTTL*
: The time to live value to use for the socket connection.

*dwReservation ID*
: Identify the bandwidth reservation to use.

### Return Values

Returns a handle to the tunnel connection or NULL if the connection cannot be made. Use **GetLastError**() to get more information when **msbdnOpenTunnel** returns NULL.

### See Also

msbdnOpenTunnel

## Data Transfer Functions

[This is preliminary documentation and subject to change.]

Content server applications send data to the MMR using TCP/IP sockets. The functions in this section handle sending data to the sockets.

| Function | Description |
|---|---|
| **msbdnSend** | Send data to the MMR |

# msbdnSend

[This is preliminary documentation and subject to change.]

The **msbdnSend** function sends data to the MMR using a tunnel or a multicast socket

```
MSBDNAPI INT WINAPI msbdnSend(
  HBDNCONN hBdnConnection,
  LPVOID lpBuffer,
  WORD wSize
);
```

**Parameters**

*hBdnConnection*
    Handle to the connection to the MMR.
*lpBuffer*
    Pointer to the data to send.
*wSize*
    Number of bytes in the buffer to send.

**Return Values**

If there is an error sending the data to the MMR, **msbdnSend** returns SOCKET_ERROR.

**See Also**

msbdnOpenTunnel

# About Output Drivers

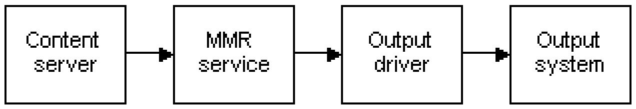
[This is preliminary documentation and subject to change.]

An output driver is a user-mode dynamic-link library (DLL) that the Microsoft Multicast Router

(MMR) requires to route data packets to a specific output device. This driver packages the data into network and hardware-specific formatting. Because this packaging is performed by the output driver and not the MMR, the MMR software does not need to be rewritten if the network or hardware changes. You can configure new multicast routers simply by changing the MMR registry entries and the output driver .dll file.

**Note** Some beta versions of the Microsoft Multicast Router call output drivers virtual interfaces (VIFs) or output subsystems.

The following diagram illustrates how a data stream flows from the content server to a broadcast output system.



Each head-end server infrastructure requires an output driver specific to its system. During testing of the MMR, Microsoft has created the following output drivers:

- Local Area Network
- Serial VBI

For information on the specific functions that must be implemented in an output driver, see Output Driver Reference.

## Local Area Network

[This is preliminary documentation and subject to change.]

The Local Area Network (LAN) output driver can be used to test broadcast server and client applications without using a satellite. Instead of forwarding data packets to particular head-end equipment, it sends them over an Ethernet LAN in the form of *Internet Protocol* (IP) *multicast* packets.

**Driver File**

Mc_out.dll

**Output Options**

The MMR can query and set the options in the following table by calling the following dynamic configuration functions for this driver: **msbdnOutputGetValue**, **msbdnOutputGetValueCount**, and **msbdnOutputSetValue**.

| Value | Type | Meaning |
|---|---|---|
| **Multicast TTL** | Dword (**DWORD**) | The *time-to-live* (TTL) value at which this driver transmits multicast packets through the output system. The default value is 1, which restricts transmission to the local network. |
| **Interface Address** | IPAddress (**IN_ADDR**) | The address of the network interface card from which this driver transmits multicast packets. If this option is not specified, the operating system uses a default interface. The interface address specified must be in Internet dotted notation (*xxx.xxx.xxx.xxx*). |

## Serial VBI

[This is preliminary documentation and subject to change.]

This output driver connects a computer serial port to a Norpak TES-3 encoder, which encodes data into the *vertical blanking interval* (VBI). This connection enables the Norpak encoder to transmit *Internet Protocol* (IP) *multicast* packets in a standard analog television signal. The Norpak encoder must be configured to use the Norpak "Bundle FEC (Forward Error Correction)" mode of transmission. The encoder must also be configured to indicate the lines of the VBI to use. The rate encoded data is sent is determined by the number of lines of the television signal that are used.

### Driver File

Vbi_out.dll

### Output Options

The MMR can query and set the options in the following table by calling the following dynamic configuration functions for this driver: **msbdnOutputGetValue**, **msbdnOutputGetValueCount**, and **msbdnOutputSetValue**.

| Value | Type | Meaning |
|---|---|---|
| **Serial Port** | String (**LPWSTR**) | The communications resource, such as the serial port COM1, to use for communication with the TES-3 encoder. |

| | | |
|---|---|---|
| **Serial Port Bit Rate** | Dword (**DWORD**) | The speed in bits per second of the serial port to use for communication with the TES-3 encoder. To locate more information on determining this speed, see Further Server Information. |
| **Debug Log Filename** | String (**LPWSTR**) | The name and path of a file to which this driver writes debugging data. |
| **Debug Log Enable** | Boolean (**BOOL**) | Turns on or off packet dump mode, causing or preventing the contents of packets to be displayed in the output log file specified by the **Debug Log Filename** option. Setting this value to TRUE turns on packet dump mode, and setting this value to FALSE turns off packet dump mode. |
| **Use Old Framing Mode** | Boolean (**BOOL**) | If set to TRUE, encodes packets in a way that is compatible with the initial experiments by Microsoft with VBI encoding. If set to FALSE, encodes packets using the Microsoft UDP/IP compression algorithm. Note that a FALSE value is not yet completely supported. |

# Output Driver Reference

[This is preliminary documentation and subject to change.]

This section documents the functions that make up the interface between the Microsoft multicast router (MMR) and an output driver, which sends data streams to a broadcast output device. Such an output driver communicates with the input unit of a broadcast data encoder. This section also documents structures that hardware vendors use to implement the output driver.

The output driver interface is documented in four groups:

- Output Driver Functions
- MMR Functions
- Output Driver Structures
- Sample Output Driver DLL

## Output Driver Functions

[This is preliminary documentation and subject to change.]

Hardware vendors should implement the output driver functions described in this section. These functions package data from a stream into a specific format, so a particular broadcast encoder can transmit the data stream over a particular network.

Hardware vendors should build their output drivers as dynamic-link libraries (DLLs) that the MMR loads at run time. An output driver DLL must implement all the functions listed following that are specified as required. Only output driver DLLs that support dynamic configuration must implement the functions listed following that are specified as optional. An output driver DLL that supports dynamic configuration enables the MMR to query or set any number of values defined by the output driver. Implementation of dynamic configuration is strongly encouraged.

Each output driver must be able to safely host multiple instances of an output system. In other words, the MMR should be able to load an output driver more than once with separate parameters and route different packet streams to such different instances of an output driver. The MMR should be able to independently start and stop each instance of an output driver; that is, the MMR should be able to load and unload a driver instance from the MMR's address space.

At run time, the MMR loads the appropriate output driver DLL with an output loader. The output loader calls the **RegQueryValueEx** function, which is part of the Microsoft® Win32® application programming interface (API). The output loader passes **RegQueryValueEx** the string OutputDLL to retrieve the file name of the output driver DLL. The output loader then calls the **LoadLibrary** Win32 function and passes this file name to retrieve a handle to the output DLL. The output loader uses this handle in calls to the **GetProcAddress** Win32 function. These calls get the address of each exported function of the output driver DLL. To locate documentation on these Win32 API functions that the output loader calls, see Further Server Information.

The following table lists and describes the output driver functions.

| **Function** | **Description** |
|---|---|
| **msbdnOutputCreate** | Creates and initializes a single instance of an output driver. This function is required. |
| **msbdnOutputDestroy** | Destroys an instance of an output driver. This function is required. |
| **msbdnOutputFreeBuffer** | Frees a buffer returned by **msbdnOutputGetValue**. This function is optional. |
| **msbdnOutputGetStatus** | Retrieves the status of an output driver. This function is required. |
| **msbdnOutputGetValue** | Retrieves a configuration value of an output driver. This function is optional. |
| **msbdnOutputGetValueCount** | Retrieves the number of configuration values supported by an output driver. This function is optional. |

| | |
|---|---|
| **msbdnOutputSendPacket** | Sends all packets of a data stream to the output system. This function is required. |
| **msbdnOutputSetValue** | Assigns a configuration value to an output driver. This function is optional. |

## msbdnOutputCreate

[This is preliminary documentation and subject to change.]

The **msbdnOutputCreate** function creates and initializes a single instance of an output driver.

```
HRESULT msbdnOutputCreate(
  OUT MSBDN_OUTPUT_SUBSYSTEM *Subsystem,
  IN DWORD ApiVersion
);
```

### Parameters

*Subsystem*
: Address of an **MSBDN_OUTPUT_SUBSYSTEM** structure that is to receive details about the output driver implementation. This function fills the structure with information about the output driver and the output device that the MMR is to send data to.

*ApiVersion*
: Highest version of the output driver interface that the caller can use. The high-order bytes specify the minor version number, for example version *x*.1. The low-order bytes specify the major version number, for example version 1.

### Return Values

Returns zero if the function was successful. If the function fails, the return value is non-zero. If this function is unable to allocate memory for an instance of the output driver, this function should return E_OUTOFMEMORY.

### Remarks

After loading an output driver DLL, the MMR immediately calls the **msbdnOutputCreate** function. This function is only called once, and no other output driver functions are called until this function returns.

The **msbdnOutputCreate** function should create any synchronization objects that it requires and try to connect to and initialize the appropriate output system or device drivers it will use. If an output driver DLL stores values in the registry when the DLL starts, **msbdnOutputCreate** should call the **RegQueryValueEx** Win32 function for each configuration value and pass the open registry key contained in the *Subsystem* parameter, the name of the configuration value to retrieve, and a buffer to hold the configuration data. The **msbdnOutputCreate** function could also provide default data for each configuration value. The **msbdnOutputCreate** function should then store retrieved or provided configuration data in values defined by the output driver. When the MMR requires a dynamic value from the output driver's configuration, the MMR calls **msbdnOutputGetValue** to retrieve such a value.

If **msbdnOutputCreate** cannot initialize an instance of the output driver, it should return a failure condition. The **msbdnOutputCreate** function should only return a failure condition if there is some condition that prevents the output driver from ever running correctly. For example, the output driver should not return an error condition if some configuration parameter prevents the hardware from functioning because the output driver can only change configuration parameters of the hardware if this function succeeds.

The **msbdnOutputCreate** function allows the MMR to specify the version of the output driver interface that a particular output driver is required to support. The function also allows the MMR to retrieve details about the output driver's implementation.

In order for the MMR to support future implementations of the output driver interface that might have functionality different from that current, a negotiation should take place in **msbdnOutputCreate**. The MMR and the particular output driver DLL indicate to each other the highest interface version that they can support. Each confirms that the other's highest version is acceptable, if it is.

To do so, the output driver DLL examines the version requested by the MMR. If this version is equal to or higher than the lowest version supported by the DLL, the call succeeds. On a successful call, the DLL returns the highest version it supports in the **Version** member of **MSBDN_OUTPUT_SUBSYSTEM**. In the structure's **Version** member, the output driver DLL returns either the highest version it supports or the version number that the MMR is requesting, whichever is lower.

After **msbdnOutputCreate** returns, the output driver DLL works under the assumption that the MMR uses the interface version returned in **Version**. If the MMR cannot work with this version, the MMR does not call any other output driver functions.

This negotiation allows both an output driver DLL and the MMR to support a range of function versions. The MMR can successfully use an output driver DLL if there is any overlap in the version ranges.

**See Also**

**msbdnOutputDestroy**, **msbdnOutputGetValue**, **MSBDN_OUTPUT_SUBSYSTEM**

## msbdnOutputDestroy

[This is preliminary documentation and subject to change.]

The **msbdnOutputDestroy** function shuts down the output system and deletes the instance of the output driver DLL before unloading the DLL.

```
HRESULT msbdnOutputDestroy(
  IN MSBDN_OUTPUT_SUBSYSTEM *Subsystem
);
```

### Parameters

*Subsystem*
Address of an **MSBDN_OUTPUT_SUBSYSTEM** structure that contains details about the output driver implementation.

### Return Values

Should always return zero for a successful status.

### Remarks

The **msbdnOutputDestroy** function is the last output driver function that the MMR calls before exiting. When **msbdnOutputDestroy** is called, the output driver DLL should stop all data transmissions, close any devices it has opened, free all packet queues, and release all resources that it allocated such as synchronization objects. The **msbdnOutputDestroy** function must complete any **PACKET_BUFFER** handles which it has queued. Because **msbdnOutputDestroy** is the last function that the MMR calls before exiting, **msbdnOutputDestroy** should block other functions' processing until it is done with all cleanup.

### See Also

**msbdnOutputCreate**, **MSBDN_OUTPUT_SUBSYSTEM**, **PacketBufferComplete**, **PACKET_BUFFER**

## msbdnOutputFreeBuffer

[This is preliminary documentation and subject to change.]

The **msbdnOutputFreeBuffer** function frees string storage that the MMR obtained from a call to the **msbdnOutputGetValue** function.

```
HRESULT msbdnOutputFreeBuffer(
  IN MSBDN_OUTPUT_SUBSYSTEM *Subsystem
  IN LPVOID Buffer
);
```

### Parameters

*Subsystem*
: Address of an **MSBDN_OUTPUT_SUBSYSTEM** structure that contains details about the output driver implementation.

*Buffer*
: Memory storage containing a null-terminated string that this function frees.

### Return Values

Should always return zero for a successful status.

### Remarks

An output driver DLL should implement **msbdnOutputFreeBuffer** so that the MMR can configure the DLL dynamically. Typically, the implementation for **msbdnOutputFreeBuffer** should free string storage by calling the **free** C function, the **delete** C++ function, or the **GlobalFree** Microsoft® Win32® base services function. The particular implementation for an output driver DLL's string memory management is left up to the hardware vendor.

### See Also

**msbdnOutputGetValue**, **MSBDN_OUTPUT_SUBSYSTEM**

## msbdnOutputGetStatus

[This is preliminary documentation and subject to change.]

The **msbdnOutputGetStatus** function retrieves the current status of the output driver DLL for the MMR.

```
HRESULT msbdnOutputGetStatus(
  IN MSBDN_OUTPUT_SUBSYSTEM *Subsystem,
  OUT HRESULT *Status
);
```

### Parameters

*Subsystem*
: Address of an **MSBDN_OUTPUT_SUBSYSTEM** structure that contains details about the output driver implementation.

*Status*
: Pointer to a value specifying the current status of the output driver DLL.

### Return Values

Returns zero if the function was successful. If the function fails, the return value is non-zero.

### Remarks

The output driver DLL should return a status of S_OK in the *Status* parameter if it is working properly, and otherwise should return an error value.

### See Also

**MSBDN_OUTPUT_SUBSYSTEM**

## msbdnOutputGetValue

[This is preliminary documentation and subject to change.]

The **msbdnOutputGetValue** function retrieves a specific dynamic value of the output driver DLL's configuration for the MMR.

```
HRESULT msbdnOutputGetValue(
  IN MSBDN_OUTPUT_SUBSYSTEM *Subsystem,
  IN/OUT MSBDN_OUTPUT_VALUE *Value
);
```

### Parameters

*Subsystem*
: Address of an **MSBDN_OUTPUT_SUBSYSTEM** structure that contains details about the output driver implementation.

*Value*
: Address of an **MSBDN_OUTPUT_VALUE** structure that is to receive details about a dynamic value of the output driver's configuration specified by the **Index** member of **MSBDN_OUTPUT_VALUE**. This function fills the structure with information about the specified value.

### Return Values

Returns zero if the function was successful. If the function fails, the return value is non-zero. If the **Index** member of **MSBDN_OUTPUT_VALUE** in the *Value* parameter is not valid, **msbdnOutputGetValue** should return E_INVALIDARG.

### Remarks

The MMR calls **msbdnOutputGetValue** to obtain a dynamic value from the output driver's configuration with the value's index. The **msbdnOutputGetValue** function should set the name and data type of the configuration value and the member of the union within **MSBDN_OUTPUT_VALUE** in the *Value* parameter corresponding to the correct data type.

If the output driver specifies the data type as a string, the output driver DLL must allocate this string on the heap. The output driver should never return a pointer to global storage; this practice is not safe in a multithreaded environment. To release memory storage for this string, the MMR calls **msbdnOutputFreeBuffer**.

**See Also**

**msbdnOutputFreeBuffer**, **msbdnOutputSetValue**, **MSBDN_OUTPUT_SUBSYSTEM**, **MSBDN_OUTPUT_VALUE**

# msbdnOutputGetValueCount

[This is preliminary documentation and subject to change.]

The **msbdnOutputGetValueCount** function retrieves the upper bound on the number of indexes of configuration values that the output driver DLL supports.

```
HRESULT msbdnOutputGetValueCount(
  IN MSBDN_OUTPUT_SUBSYSTEM *Subsystem,
  OUT DWORD *ValueCount
);
```

**Parameters**

*Subsystem*
: Address of an **MSBDN_OUTPUT_SUBSYSTEM** structure that contains details about the output driver implementation.

*ValueCount*
: Address of a value that specifies the upper bound on the number of indexes of configuration values that the output driver DLL supports. Usually, this value is a constant.

**Return Values**

Should always return zero for a successful status.

**Remarks**

The MMR calls **msbdnOutputGetValueCount** to obtain the number of dynamic configuration parameters the output system might support. For example, if an output system supports a serial-port value, a bit-rate value, and a framing-mode value, **msbdnOutputGetValueCount** should store the number three in the memory to which the *ValueCount* parameter points.

**See Also**

**msbdnOutputGetValue**, **MSBDN_OUTPUT_SUBSYSTEM**

# msbdnOutputSendPacket

[This is preliminary documentation and subject to change.]

The **msbdnOutputSendPacket** function sends all packets of a data stream to the output system.

```
HRESULT msbdnOutputSendPacket(
  IN MSBDN_OUTPUT_SUBSYSTEM *Subsystem,
  IN PACKET_BUFFER *Packet
);
```

**Parameters**

*Subsystem*
Address of an **MSBDN_OUTPUT_SUBSYSTEM** structure that contains details about the output driver implementation.

*Packet*
Address of a **PACKET_BUFFER** structure that describes a packet of data.

**Return Values**

Returns zero if the output driver successfully transmitted the packet. If the function fails, the return value is non-zero.

**Remarks**

While a stream is active, the stream functionality of the MMR calls the **msbdnOutputSendPacket** function in the output driver DLL. The MMR does so to send a stream's data packets to the broadcast encoder or to other output systems. The **msbdnOutputSendPacket** function is responsible for translating protocol types, generating network-specific addresses, and reformatting data to follow network-specific protocols.

The output driver can alter the **Start** and **End** members of a **PACKET_BUFFER** structure and can alter the memory to which the **Data** member of **PACKET_BUFFER** points. However, the output driver must not alter the information within the **Data** member itself. The MMR stores the actual data that the output driver transmits in the buffer to which the **Data** member of **PACKET_BUFFER** points. The data that the output driver transmits begins at the index given by the **Start** member of

**PACKET_BUFFER** and ends at the index given by the **End** member of **PACKET_BUFFER**. The buffer for transmitted packets is structured in this manner to allow output drivers to efficiently add or remove packet headers and trailers without requiring that the output driver copy the entire packet elsewhere in memory.

The length of the actual packet data is determined by subtracting the **PACKET_BUFFER**'s **Start** member from its **End** member. Provided that there is sufficient space at the beginning of the packet buffer, the output driver could decrease the value of the **PACKET_BUFFER**'s **Start** member and then store new data in that memory location. The output driver could also extend the packet by increasing the **PACKET_BUFFER**'s **End** member. However, the output driver must never increase the value of the **PACKET_BUFFER**'s **End** member beyond its **Max** member. The output driver can also change the values of the **PACKET_BUFFER**'s **Start** and **End** members to decrease the length of the packet.

Examples of the output driver manipulating packets include:

- Removing an IP and then a UDP header in order to extract just the UDP body data
- Encapsulating an IP packet in a larger frame

The output driver can accomplish these tasks without copying the packet data, which facilitates high-speed applications.

The output driver must "complete" the packet buffer when it is done processing it. If the output driver can finish processing the packet immediately, it should complete the packet buffer before returning from **msbdnOutputSendPacket**; this operation is synchronous. Otherwise, the output driver might store the packet in a queue and complete the packet later in a different thread; this operation is asynchronous.

If the MMR receives data with protocol types it cannot decipher, it forwards the data to the output driver without modification.

**See Also**

**MSBDN_OUTPUT_SUBSYSTEM**, **PacketBufferComplete**, **PACKET_BUFFER**

# msbdnOutputSetValue

[This is preliminary documentation and subject to change.]

The **msbdnOutputSetValue** function assigns a dynamic value to an output driver DLL's configuration specified by the MMR.

```
HRESULT msbdnOutputSetValue(
  IN MSBDN_OUTPUT_SUBSYSTEM *Subsystem,
  IN MSBDN_OUTPUT_VALUE *Value
);
```

### Parameters

*Subsystem*
Address of an **MSBDN_OUTPUT_SUBSYSTEM** structure that contains details about the output driver implementation.

*Value*
Address of an **MSBDN_OUTPUT_VALUE** structure. This structure contains details about a dynamic value for the output system's configuration that the MMR requires set.

### Return Values

Returns zero if the function was successful. If the function fails, the return value is non-zero. If the **Index** member of **MSBDN_OUTPUT_VALUE** in the *Value* parameter is not valid or the **Type** member is not correct, **msbdnOutputGetValue** should return E_INVALIDARG.

### Remarks

The MMR calls **msbdnOutputSetValue** to set a specific dynamic configuration value. The index of the value is stored in the **Index** member of **MSBDN_OUTPUT_VALUE** in the *Value* parameter. The output driver DLL should look up the dynamic configuration value corresponding to this index, check to make sure that the data type passed by the MMR is correct for this value, set its own internal value for this configuration value, and should store this configuration value in persistent storage. An output driver usually stores values in the registry. If this configuration value affects communication with external equipment, such as setting a bitrate divisor, this change should take place immediately, provided doing so does not cause other problems.

If an output driver stores values in the registry, **msbdnOutputSetValue** should call the **RegSetValueEx** Win32 function and pass the open registry key contained in the *Subsystem* parameter, the name of the configuration value to set, and the buffer containing the configuration data. The buffer that holds this configuration data contains the address of the union member of **MSBDN_OUTPUT_VALUE** in the *Value* parameter.

### See Also

**msbdnOutputGetValue**, **MSBDN_OUTPUT_SUBSYSTEM**, **MSBDN_OUTPUT_VALUE**

## MMR Functions

[This is preliminary documentation and subject to change.]

The Microsoft Multicast Router (MMR) also is a dynamic-link library (DLL), which output driver DLLs bind to at run time. In this library, the MMR exports a callback function that allows output driver DLLs to report internal events and a completion function that allows the output driver to report that it has finished with a packet.

Hardware vendors should implement their output driver code by including the Bridge.h header file and by calling the MMR functions described in this section. When vendors build output driver DLLs, they should link their output driver code to the Bdnapi.lib library.

The following table lists and describes the MMR functions.

| Function | Description |
|---|---|
| **msbdnBridgeReportEvent** | Informs the MMR to record output driver events directly in the system event log. |
| **PacketBufferComplete** | Informs the MMR that the output driver is done processing a packet. |

# msbdnBridgeReportEvent

[This is preliminary documentation and subject to change.]

The **msbdnBridgeReportEvent** function informs the MMR to record output driver events directly in the system event log.

```
HRESULT msbdnBridgeReportEvent(
  IN MSBDN_OUTPUT_SUBSYSTEM *Subsystem,
  IN WORD wType,
  IN DWORD dwErrorCode,
  IN LPCSTR szMessage
);
```

### Parameters

*Subsystem*
: Address of an **MSBDN_OUTPUT_SUBSYSTEM** structure that contains details about the output driver implementation.

*wType*
: A value that can be one of the following three standard Win32 error types: EVENTLOG_INFORMATION_TYPE, EVENTLOG_ERROR_TYPE, or EVENTLOG_WARNING_TYPE.

*dwErrorCode*
: A value that can be a Win32 error code, a Windows Sockets error code, a standard system HRESULT value, or one of status codes defined in the Brerror.h header file. If this value is an error code, then the full textual description of the error message will be included with the event in the *szMessage* parameter.

*szMessage*

A Unicode string containing the message to record in the event log.

**Return Values**

Returns zero if the function was successful. If the function fails, the return value is non-zero. To get extended error information, call the Win32 function **GetLastError**. To locate more information on **GetLastError**, see Further General Information.

**Remarks**

The **msbdnBridgeReportEvent** function allows an output driver to easily record events in the event log. An output driver can also call the functions of the underlying Win32 event logging application programming interface (API). Using **msbdnBridgeReportEvent** reduces overhead for an output driver implementation.

The MMR implements **msbdnBridgeReportEvent** as an inline function in the Bridge.h header file. The **msbdnBridgeReportEvent** implementation calls the **ReportEvent** member of the **MSBDN_BRIDGE_CALLBACKS** structure for a specific output driver instance defined by an **MSBDN_OUTPUT_SUBSYSTEM** structure. An inline function by definition is a function whose code gets substituted in place of the actual call to that function. That is, whenever the compiler encounters a call to that function, it merely replaces it with the code itself, thereby saving overhead.

**See Also**

**MSBDN_BRIDGE_CALLBACKS**, **MSBDN_OUTPUT_SUBSYSTEM**

# PacketBufferComplete

[This is preliminary documentation and subject to change.]

The **PacketBufferComplete** function informs the MMR that the output driver is done processing a packet and provides status on the completed packet.

```
void PacketBufferComplete(
  IN PACKET_BUFFER *Packet,
  IN DWORD dwStatus
);
```

**Parameters**

*Packet*
Address of a **PACKET_BUFFER** structure that describes a packet the output driver is finished with.

*dwStatus*
Value specifying the status of the completed packet. The following table lists and describes the

valid status values.

| **Value** | **Meaning** |
|---|---|
| PACKET_COMPLETE_FAILURE | A failure occurred in processing the packet. |
| PACKET_COMPLETE_NO_ROUTE | The output driver was unable to route the packet. |
| PACKET_COMPLETE_OVERFLOW | The output driver cannot process the packet synchronously and lacks queue space to process the packet asynchronously. |
| PACKET_COMPLETE_SUCCESS | Processing of the packet completed successfully. |

**Return Values**

None

**Remarks**

The MMR implements **PacketBufferComplete** as an inline function in the Packet.h header file; the Bridge.h header file includes Packet.h. The **PacketBufferComplete** implementation calls the **CompletionFunc** member of the **PACKET_BUFFER** structure for the passed in packet.

Output driver DLLs can be implemented for particular output systems that use first-in-first-out (FIFO) chips of a fixed-length in the range of 32 to 64 slots. If such an output driver fills the output system's FIFO before transmitting all the packet data, then the output driver must inform the MMR that it is done processing the packet with a status value of PACKET_COMPLETE_OVERFLOW.

**See Also**

**PACKET_BUFFER**

## Output Driver Structures

[This is preliminary documentation and subject to change.]

To write output drivers that format and send data to a broadcast encoder or other output systems, you can use the structures discussed in this section. These structures describe specific output systems and drivers and the data packets that output drivers send to their systems. The following table lists and describes these structures.

| Structure | Description |
|---|---|
| MSBDN_BRIDGE_CALLBACKS | Contains information about pointers to callback functions exported by the MMR |
| MSBDN_OUTPUT_SUBSYSTEM | Contains information representing a specific output system |
| MSBDN_OUTPUT_VALUE | Contains information about a configuration value |
| PACKET_BUFFER | Contains information about a packet of data that an output driver transmits |

## MSBDN_BRIDGE_CALLBACKS

[This is preliminary documentation and subject to change.]

The **MSBDN_BRIDGE_CALLBACKS** structure contains information about pointers to callback functions exported by the MMR that allow output driver DLLs to report internal status.

```
typedef struct MSBDN_BRIDGE_CALLBACKS {
    DWORD Version;
    HRESULT     (*ReportState) (struct MSBDN_OUTPUT_SUBSYSTEM *,
                                DWORD state, LPCSTR message);
    HRESULT     (*ReportActivity) (struct MSBDN_OUTPUT_SUBSYSTEM *,
                                WORD type, DWORD amount);
    HRESULT     (*ReportEvent) (struct MSBDN_OUTPUT_SUBSYSTEM *,
                                WORD, DWORD, LPCWSTR);
} MSBDN_BRIDGE_CALLBACKS;
```

### Members

**Version**
Version number of the MMR callback reference.

**ReportState**
Pointer to a function that an output driver calls to inform the MMR about the state of such output driver.

**ReportActivity**
Pointer to a function that an output driver calls to inform the MMR about real-time bandwidth usage and minor problems with the output system.

**ReportEvent**
Pointer to a function that an output driver calls to inform the MMR of an output driver event.

### See Also

msbdnBridgeReportEvent

# MSBDN_OUTPUT_SUBSYSTEM

[This is preliminary documentation and subject to change.]

The **MSBDN_OUTPUT_SUBSYSTEM** structure contains information representing a specific output system that an output driver can configure dynamically and to which an output driver sends data packets.

```
typedef struct MSBDN_OUTPUT_SUBSYSTEM {
    DWORD                  Version;
    MSBDN_BRIDGE_CALLBACKS BridgeCallbacks;
    MSBDN_SUBSYSTEM_ID     OutputSubsystemID;
    HKEY                   RegistryKey;
    LPVOID                 DriverContext;
} MSBDN_OUTPUT_SUBSYSTEM;
```

## Members

**Version**
Version of the output driver interface that the output driver requires the MMR to support.

**BridgeCallbacks**
An **MSBDN_BRIDGE_CALLBACKS** structure containing information about pointers to callback functions exported by the MMR that allow output driver DLLs to report internal status.

**OutputSubsystemID**
A value that identifies a particular output system. This value uniquely identifies an instance of an output driver DLL. An **MSBDN_SUBSYSTEM_ID** data type is defined as a **DWORD** data type.

**RegistryKey**
The handle of an open registry key that the output driver can use to retrieve any previously stored configuration data and to store new configuration data. The output driver can use this registry key handle but should not close or alter it. This value should not be NULL.

**DriverContext**
Reserved for the output driver to use. Typically, this member points to a structure defined by the output driver DLL that contains all of the internal state information of an instance of an output system such as configuration data, synchronization objects, and device handles.

## Remarks

In every output driver function call, the MMR passes a pointer to **MSBDN_OUTPUT_SUBSYSTEM** in order to uniquely identify a specific instance of an output driver DLL.

**See Also**

MSBDN_BRIDGE_CALLBACKS

# MSBDN_OUTPUT_VALUE

[This is preliminary documentation and subject to change.]

The **MSBDN_OUTPUT_VALUE** structure contains information about a dynamic value for an output driver's configuration.

```
typedef struct MSBDN_OUTPUT_VALUE {
    DWORD   Index;
    DWORD   Type;
    LPWSTR Name;
    union {
        DWORD    Dword;
        LPWSTR   String;
        IN_ADDR IPAddress;
        BOOL     Boolean;
    };
} MSBDN_OUTPUT_VALUE;
```

**Members**

**Index**
Index number of a dynamic value for an output driver's configuration.

**Type**
Value specifying the type of information stored as configuration data. The following table lists and describes the valid values.

| Value | Meaning |
|---|---|
| MSBDN_OUTPUT_VALUE_BOOLEAN | **BOOL** data type |
| MSBDN_OUTPUT_VALUE_DWORD | **DWORD** data type |
| MSBDN_OUTPUT_VALUE_IPADDRESS | **IN_ADDR** data type |
| MSBDN_OUTPUT_VALUE_STRING | **LPWSTR** data type |

**Name**
A null-terminated string containing the name of a dynamic value for an output driver's configuration.

**Dword**
Member of the union contained in **MSBDN_OUTPUT_VALUE** that can hold a 32-bit number (**DWORD**) as configuration data.

**String**

Member of the union contained in **MSBDN_OUTPUT_VALUE** that can hold a Unicode string (**LPWSTR**) as configuration data.

**IPAddress**

Member of the union contained in **MSBDN_OUTPUT_VALUE** that can hold an IP address (**IN_ADDR**) as configuration data.

**Boolean**

Member of the union contained in **MSBDN_OUTPUT_VALUE** that can hold a Boolean value (**BOOL**) as configuration data.

### See Also

**msbdnOutputGetValue**, **msbdnOutputSetValue**

## PACKET_BUFFER

[This is preliminary documentation and subject to change.]

The **PACKET_BUFFER** structure contains information about a packet of data that an output driver sends to its output system.

```
typedef struct PACKET_BUFFER {
    LPBYTE                 Data;     // storage for packet data
    DWORD                  Start;    // where the data begins
    DWORD                  End;      // where the data ends
    DWORD                  Max;      // the physical length of Data
    DWORD                  Context;
    WORD                   Protocol;
    WORD                   AddressLength;
    BYTE                   Address[16];
    PACKET_COMPLETION_FUNC CompletionFunc;
} PACKET_BUFFER;
```

### Members

**Data**

Buffer containing the data that the output driver transmits.

**Start**

The beginning memory location of the transmitted data.

**End**

The ending memory location of the transmitted data.

**Max**

The maximum physical length of the transmitted data. The **End** member can never be increased beyond the value of **Max**.

**Context**

Value specifying the context of the transmitted data.

**Protocol**

Value specifying the format of the data. The following table lists and describes the valid format values.

| **Value** | **Meaning** |
|---|---|
| PACKET_BUFFER_PROTOCOL_IP | The packet body is a full IP packet, including header and body. Output drivers should extract an IP address from the message body. The **Address** member of **PACKET_BUFFER** is not used. |
| PACKET_BUFFER_PROTOCOL_VBI_RAW | The packet is a VBI frame. The **Address** member of **PACKET_BUFFER** is not used. |

**AddressLength**
Size, in bytes, of the **Address** array. If **Address** is null, **AddressLength** is zero.

**Address**
An array of bytes specifying an address to which the packet belongs. This array can be no longer than 16 bytes.

**CompletionFunc**
Pointer to a function that an output driver DLL calls to inform the MMR that the DLL is done processing the packet buffer. In this call, the DLL provides status on the completed packet.

**Remarks**

The MMR keeps track of packets with the **PACKET_BUFFER** structure. An output driver uses **PACKET_BUFFER** to route packets from the MMR to a specific output system.

The **PACKET_BUFFER** structure provides output drivers with an easy way to add or remove packet headers without requiring memory allocation and copy operations. For example, to remove an IP header from a packet, advance the **Start** member of **PACKET_BUFFER** by the length of the IP header. To add an Ethernet frame header to the front of a packet, first insure that there is sufficient space at the beginning of the packet buffer by checking to make sure that the **Start** member is greater than or equal to the length of an Ethernet header. Then, subtract the length of the Ethernet header from the **Start** member and fill in the appropriate members in the Ethernet header.

When the MMR calls the **msbdnOutputSendPacket** function on a specific instance of an output driver, such an output driver takes responsibility for the passed **PACKET_BUFFER** while processing it. This output driver must eventually call the **PacketBufferComplete** function to indicate to the MMR that the output driver is done processing.

**See Also**

**msbdnOutputSendPacket**, **PacketBufferComplete**

# Sample Output Driver DLL

[This is preliminary documentation and subject to change.]

This section describes source code for the sample output driver dynamic-link libraries (DLLs) that are provided as part of the beta program for the Microsoft® Windows® 98 operating system and the Platform Software Development Kit (SDK) to give hardware vendors assistance in writing their own output driver DLLs. To be of maximum value, this section requires the user to have the Main.cpp source-code files under the Mc_out and Nulloss directories close at hand.

**Note** Output driver DLLs are components of a broadcast server and therefore only run on Microsoft® Windows NT® Server version 4.0, or newer, operating system.

The following sections describe:

- Sample Output Driver Overview
- Sample Output Driver Anatomy
- Sample Output Driver Walk-through

## Sample Output Driver Overview

[This is preliminary documentation and subject to change.]

The beta program for the Microsoft® Windows® 98 operating system and the Platform SDK provide the source code for two sample output drivers described as follows.

The source code for the Mc_out.dll sample demonstrates an output driver that broadcasts over a local area network (LAN). Such an output driver can be used to test broadcast server and client applications without using a satellite or a *vertical blanking interval* (VBI) transmitter. Instead of forwarding data packets from the Microsoft Multicast Router (MMR) to particular head-end equipment, Mc_out.dll sends them over an Ethernet LAN in the form of *Internet Protocol* (IP) *multicast* packets. The source code for the Mc_out.dll shows implementations for the optional functions that support dynamic configuration in addition to the required functions. The required and optional functions for an output driver are described in Sample Output Driver Anatomy.

The source code for the Nulloss.dll sample demonstrates an output driver that is useful for debugging purposes. Such an output driver can collect packets from the MMR, and rather than forwarding those packets to particular head-end equipment, periodically display the bandwidth of the packets. Because the Nulloss.dll sample does not have any configuration values, it does not export the optional functions that support dynamic configuration and its source code does not show implementations for those functions.

## Sample Output Driver Anatomy

[This is preliminary documentation and subject to change.]

After installing the Broadcast Architecture Programmer's Reference material, the Main.cpp files for both samples are available. These sample files aid the user in understanding the purpose of each sample and its relationship to the MMR. The following table shows how the MMR calls and the uses of the exported output driver functions. Documentation for the following exported output driver functions can be found in the Output Driver Functions section.

| Function | Description |
|---|---|
| `HRESULT msbdnOutputCreate( MSBDN_OUTPUT_SUBSYSTEM * subsystem, DWORD version);` | Creates and initializes a single instance of an output driver. This function is required. |
| `HRESULT msbdnOutputDestroy( MSBDN_OUTPUT_SUBSYSTEM * subsystem);` | Destroys an instance of an output driver. This function is required. |
| `HRESULT msbdnOutputFreeBuffer( MSBDN_OUTPUT_SUBSYSTEM * subsystem, LPVOID buffer);` | Frees a buffer returned by **msbdnOutput GetValue**. This function is optional. |
| `HRESULT msbdnOutputGetStatus( MSBDN_OUTPUT_SUBSYSTEM * subsystem, HRESULT * status);` | Retrieves the status of an output driver. This function is required. |
| `HRESULT msbdnOutputGetValue( MSBDN_OUTPUT_SUBSYSTEM * subsystem, MSBDN_OUTPUT_VALUE * value);` | Retrieves a configuration value of an output driver. This function is optional. |
| `HRESULT msbdnOutputGetValueCount( MSBDN_OUTPUT_SUBSYSTEM * subsystem, DWORD * count);` | Retrieves the number of configuration values supported by an output driver. This function is optional. |
| `HRESULT msbdnOutputSendPacket( MSBDN_OUTPUT_SUBSYSTEM * subsystem, PACKET_BUFFER * packet);` | Sends all packets of a data stream to the output system. This function is required. |
| `HRESULT msbdnOutputSetValue( MSBDN_OUTPUT_SUBSYSTEM * subsystem, MSBDN_OUTPUT_VALUE * value);` | Assigns a configuration value to an output driver. This function is optional. |

## Sample Output Driver Walk-through

[This is preliminary documentation and subject to change.]

The following steps briefly describe creation, use, and destruction for the Mc_out.dll sample. The Mc_out.dll functions mentioned in this section are described in greater detail in Output Driver Functions.

1. At run time, the MMR loads Mc_out.dll with an output loader and retrieves the address of each exported function of the DLL. The MMR then calls the **msbdnOutputCreate** function and specifies the version of the required output system so it can retrieve details of the specific output driver implementation. The **msbdnOutputCreate** function then connects to the appropriate output system with specific options obtained from the output driver's configuration and configures the output system. The **msbdnOutputCreate** function calls the WinSock **socket** function to create a socket that uses UDP for the Internet address family and then calls the WinSock **bind** function to bind a local address to such a socket. If the socket could not be bound to a UDP/IP port, **msbdnOutputCreate** calls the **msbdnBridgeReportEvent** function and passes the EVENTLOG_ERROR_TYPE error type and the error value provided by WinSock to inform the MMR to record the error directly in the system event log.
2. When the MMR requires dynamic values of the output driver's configuration, it calls the **msbdnOutputGetValueCount** and **msbdnOutputGetValue** functions.
3. When the MMR assigns a dynamic value to an output driver's configuration, it calls the **msbdnOutputSetValue** function.
4. When the MMR has data to send, it calls the **msbdnOutputSendPacket** function to send a packet of a data stream to the output system. After **msbdnOutputSendPacket** has finished processing the packet, it informs the MMR by calling the **PacketBufferComplete** function and provides status on the completed packet.
5. Before exiting, the MMR calls the **msbdnOutputDestroy** function to stop all data transmissions and free all resources that the output driver allocated.

# Internet Channel Broadcast Server

[This is preliminary documentation and subject to change.]

Internet channel broadcasting refers to an architecture that enables World Wide Web sites to be collected, packaged, and then broadcast to multiple subscribers simultaneously.

Currently, the Web operates on a one-to-one basis. Each user who wants to view a site must create a separate connection to that site. If too many users try to access a site at the same time, the server is unable to handle all of the requests. In contrast, Internet channel broadcasting operates on a one-to-many basis. Web sites are broadcast to many users at once and stored in each user's cache until the user is ready to view them.

For more information, see the following sections:

## System Requirements

[This is preliminary documentation and subject to change.]

The Internet Channel Broadcast server software runs on the Microsoft® Windows NT® operating system, version 4.0 or greater. In addition, the server requires that the following software components be installed:

- Service Pack 3 for version 4.0 of the Microsoft® Windows NT® operating system
- Microsoft® Access 97 or Data Access Objects (DAO) version 3.5
- Microsoft® Internet Explorer version 4.0

## Overview of Internet Channel Broadcasting

[This is preliminary documentation and subject to change.]

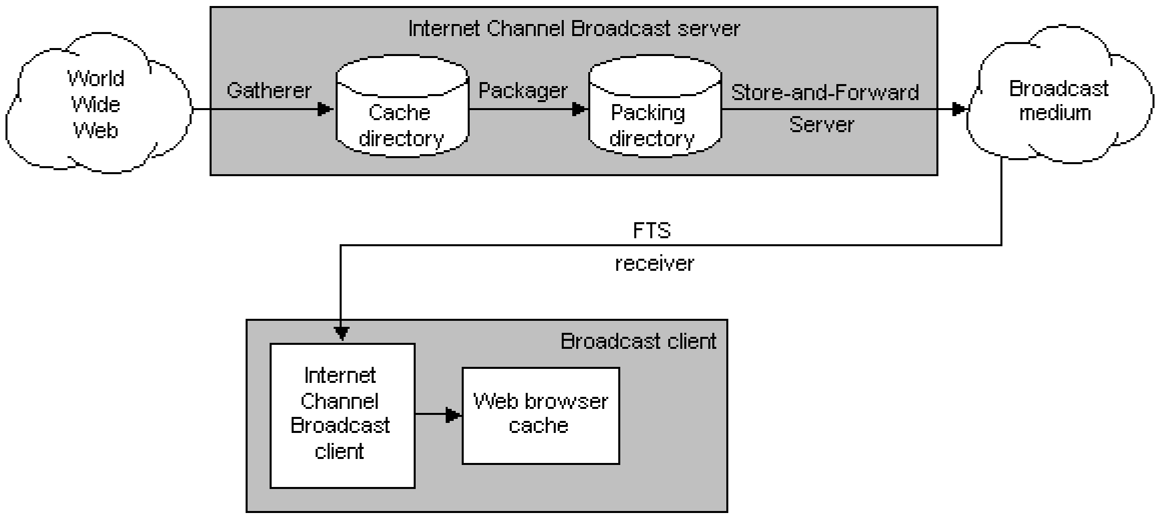
There are two parts to the Internet channel broadcasting system:

- Internet Channel Broadcast server
- Internet Channel Broadcast client

The server is administered by a *service provider* that supports Internet channel broadcasting. This server gathers files from the Web *channels* specified in the Scheduler Database and caches these files. The server then packages the files for transmission. The packaged files are transmitted using a broadcast medium, for example as a signal coded into the *vertical blanking interval* (VBI) of an analog television broadcast, or as packets over a local area network (LAN).

The Internet Channel Broadcast client runs on the user's computer. It decodes and filters broadcasts from the Internet Channel Broadcast server. When a Web site that the user has subscribed to is transmitted, the client unpacks the transmission into the Web browser cache.

The following illustration provides an overview of the Internet channel broadcasting architecture.



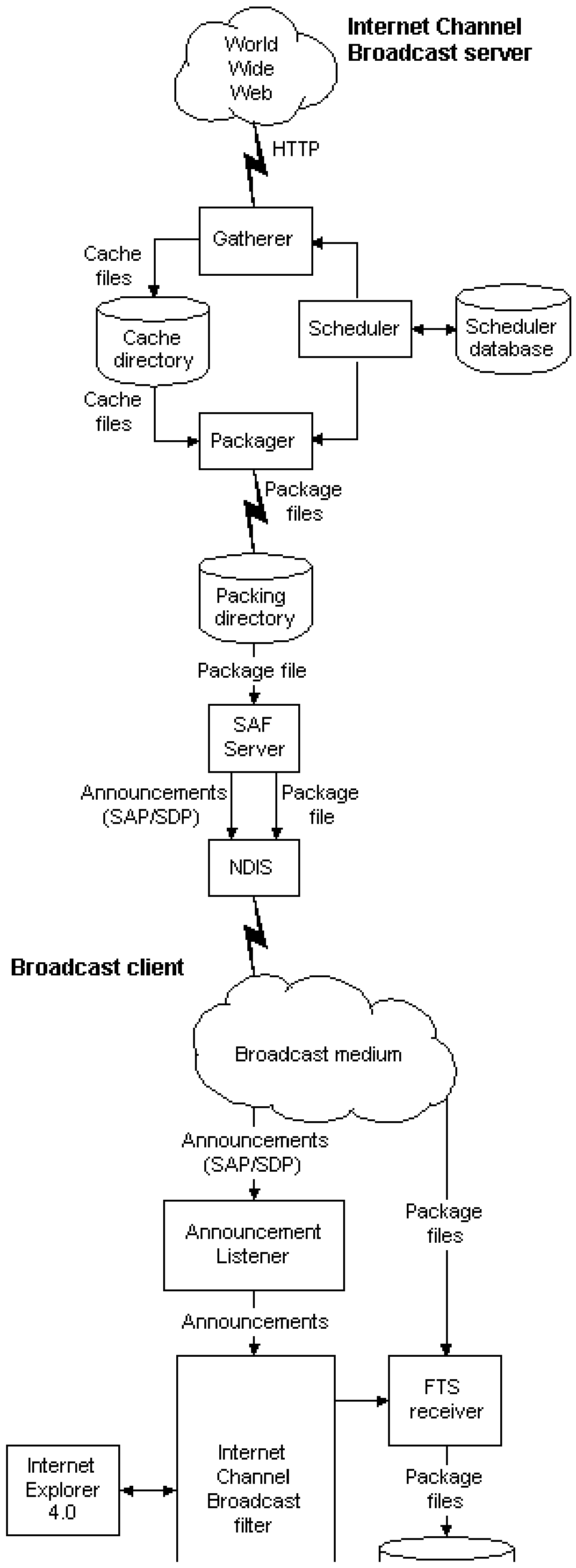
At the server end, a Gatherer component collects files from the Web and stores them in a cache directory. The Packager component packages files from that directory and stores the packages in the package directory. These packages are then transmitted to the client over the broadcast medium. When the client receives the packages, it unpackages and reconstructs them into the browser cache.

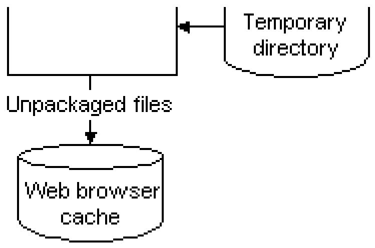
For a more details on the Internet channel broadcasting system, see Internet Channel Broadcasting Architecture, Internet Channel Broadcast Client, and Internet Channel Broadcast Server Architecture.

## Internet Channel Broadcasting Architecture

[This is preliminary documentation and subject to change.]

The following illustration shows, in detail, the flow of data in a Internet channel broadcasting system from the Web to a user's computer.





For more information, see Internet Channel Broadcast Client and Internet Channel Broadcast Server Architecture.

## Internet Channel Broadcast Server Architecture

[This is preliminary documentation and subject to change.]

The Internet Channel Broadcast server is administered by a *service provider*. This server gathers files from specified Web sites, caches them, packages them for transmission, and stores them. The packaged files are then broadcast using the fault-tolerant, one-way *File Transfer Service* (FTS) protocol. The files can be broadcast over any medium that supports *Internet Protocol* (IP) *multicast*. These media currently include local area networks (LANs), for which files are broadcast as packets, and analog television, for which files are encoded in the *vertical blanking interval* (VBI).

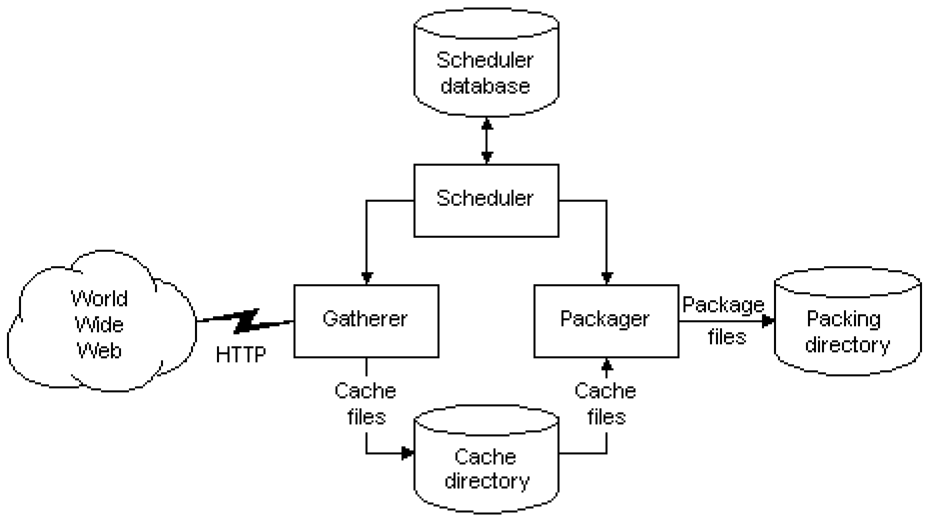
The Internet Channel Broadcast Server can be broken down into two main areas of functionality:

- Collection and packaging of files
- Storage and transmission of file packages

The Scheduler, Gatherer, Cache Server, and Packager collect and package files. The Store-and-Forward (SAF) Server stores and transmits file packages.

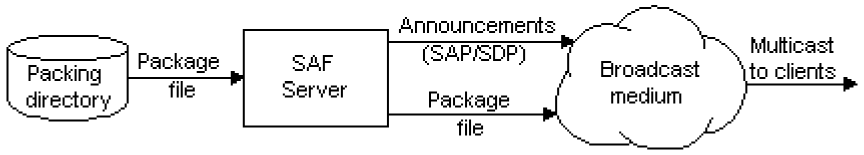
The server process goes as follows. The Scheduler checks the entries in the Scheduler database. When a file *channel* is ready to be collected, the Scheduler calls the Gatherer to get that channel from the Web. The Gatherer checks the files already in the cache against those on the Web and updates only those that have changed. The Cache Server manages the files in the cache. After all the updated files for a channel have been gathered, the Packager packages the channel files. When packaging is complete, the Packager stores the packaged files in the package directory.

The following illustration shows how the various server components work together to collect and package channels of files from the Web.



The SAF Server uses a simple round-robin mechanism to decide when packaged channels should be broadcast to clients. When all of the currently packaged channels have been broadcast, the SAF Server starts again at the first channel and rebroadcasts the channels. In a future version of Internet channel broadcasting, users will be able to specify times for package broadcasts by using the Scheduler database.

The SAF Server generates announcements for the packaged channels in the package store and broadcasts the packages to clients over the broadcast medium. The following illustration shows how the SAF Server moves packaged files from the package directory to broadcast medium.



## Components and Options for the Internet Channel Broadcast Server

[This is preliminary documentation and subject to change.]

There are five components in the Internet Channel Broadcast server architecture:

- Scheduler, which manages the Gatherer and Packager.
- Gatherer, which collects specified sites from the Web.
- Cache Server, which manages the server directory storing those sites.
- Packager, which combines site files into the package files that are broadcast.
- Store-and-Forward Server, which *multicasts* package files.

The Scheduler is a utility with a graphical interface. The other four modules run at the command prompt and make up the core of the server technology.

For more information, see the following topics:

- Internet Channel Broadcast Server Architecture
- Scheduler Database
- Scheduler Database Schema
- Internet Channel Broadcast Server Manager
- System Options for Internet Channel Broadcasting
- Using Internet Channel Broadcast Server

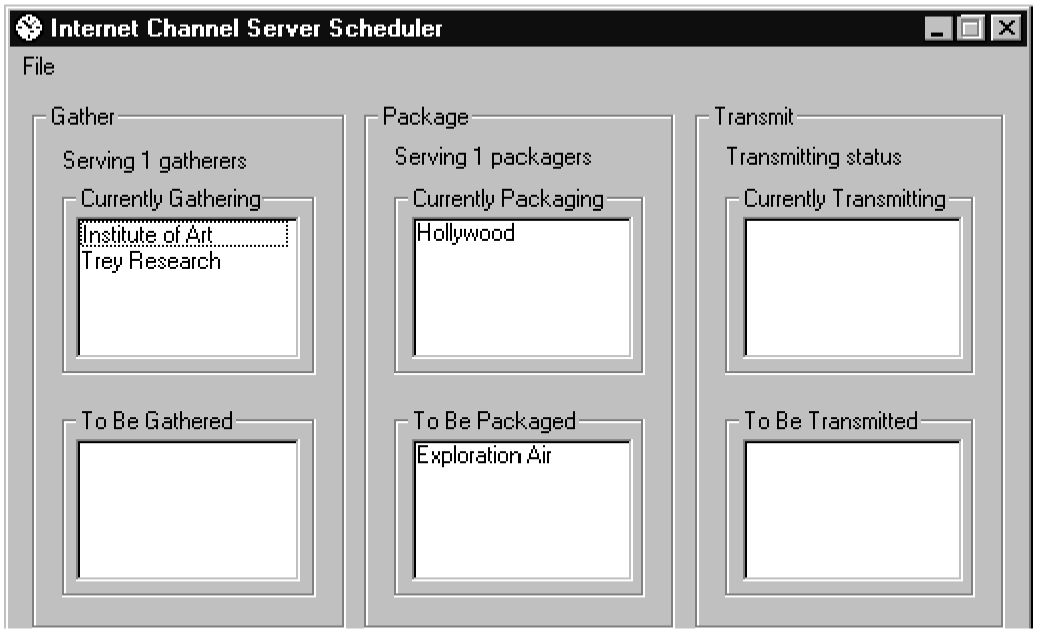
## Scheduler

[This is preliminary documentation and subject to change.]

The Scheduler, Wbcsched.exe, is a component that coordinates the Gatherer and Packager to collect and package *channels* of files from the Web. It uses the Scheduler database, Groupdb.mdb, to store information about these file channels and how often they should be collected.

The Scheduler checks the records in the Scheduler database to find channels that need to be collected. When a channel is ready to be collected, the Scheduler calls the Gatherer to retrieve the files. Once the files have been gathered, the Scheduler calls the Packager to package the channel.

The Scheduler window displays which files are currently being or are scheduled to be gathered, packaged, or transmitted. Watching the Scheduler, you can follow a channel's progress through the Internet channel broadcasting system.



## Scheduler Database

[This is preliminary documentation and subject to change.]

The Scheduler uses a database, Groupdb.mdb, to store information about the channels that the Internet Channel Broadcast server broadcasts and about how often they should be collected and transmitted. The Internet channel broadcasting options stored in the Scheduler database are described in Scheduler Database Schema.

You can create and update channel records in the Scheduler database using Microsoft® Access. For more information, see Specifying Channels to Broadcast.

After a Web site is entered into the Scheduler database, the gathering, packaging, and forwarding of the site is handled automatically by other components of the Internet Channel Broadcast server.

### Scheduler Database Schema

[This is preliminary documentation and subject to change.]

The following channel options are stored in the fields of the Scheduler database. You can add or delete channels and edit channel broadcasting options in the database using Microsoft Access 97.

| **Field** | **Description** |
|---|---|
| Title | A unique name for the Internet channel. |
| Package Type | An enumerated value that specifies the type of content being broadcast. Typically, this is set to CDF_CHANNEL. For a complete list of supported values, see Package Types. |
| URL Spec | The Uniform Resource Locator (URL) that points to the Channel Definition Format file that defines this channel. |
| DoCollect | A value that indicates whether the Gatherer collects this channel. |

| | |
|---|---|
| NextCollect | The next time the Gatherer will collect this channel. The time is specified in the format *mm*/*dd*/*yy hh*:*min* where *mm* is month, *dd* day, *yy* year, *hh* hour, and *min* minute. Deleting the value in this field causes the channel to be gathered immediately. |
| LastCollect | The last date and time the channel was gathered. |
| CollectPeriodSeconds | The interval, in seconds, between the times at which the channel should be gathered. The default is that the channel is gathered every 24 hours, or 86,400 seconds. |
| DoTransmit | A value that indicates whether the Packager should package this channel. |
| NextTransmit | The next time the Store-and-Forward Server will transmit this channel. The time is specified in the format noted previously. Deleting the value in this field causes the channel to be transmitted immediately. |
| LastTransmit | The last date and time the channel was transmitted. |
| TransmitPeriodSeconds | The interval, in seconds, between the times at which the channel should be transmitted. The default is that the channel is gathered every 24 hours, or 86,400 seconds. |
| ObjInfo | Additional information about the channel, typed in the format *key***:***value* where each key-value pair is separated from the next by a newline character. Currently, the only supported key is **dest-dir**.<br><br>This key is used to transmit files that are stored in the client's file system instead of the browser cache. The **dest-dir** key should be set for files that are not appropriate for the browser cache, for example, files that the content provider wants to remain persistent on the client.<br><br>The **dest-dir** key specifies a subdirectory under the directory specified in the **recv_dir** registry key of the client computer. The Web site files will be unpackaged and stored in this subdirectory.<br><br>Note that the Internet Channel Broadcasting filter on the client prevents does not allow |

| | |
|---|---|
| | servers to use specify destination directories that are higher than \Recv in the directory structure. In other words, you cannot use specify a destination directory such as, ..\..\..\Windows\System, in order to place files in C:\Windows\System. Only \Recv and its subdirectories are valid destination directories. |
| MaxSize | The size, in kilobytes, of the storage space allowed in the cache directory for this channel. If this value is exceeded while the channel is being gathered, any remaining files are not collected. |
| Size | The total size, in kilobytes, of storage space required to store the files gathered for this channel. |
| Status | A value that indicates whether the Gatherer successfully retrieved all of the channel files on its last attempt. The possible values are Failed and Success. |
| DeltaLevel | This field is reserved. |
| CDFImage | The name of the .gif file for the icon to display for this channel. |

## Package Types

[This is preliminary documentation and subject to change.]

The Package Type field in the Scheduler database schema describes the type of content being broadcast for a particular channel. The package type is passed to the Internet Channel Broadcast client in the announcement header. The client uses the package type information to determine how to unpackage the files and their storage location. The following table lists and describes possible package types.

| **Package type** | **Description** |
|---|---|
| AD | Rotating advertisements. This package type is reserved for use by Broadcast Architecture to update the rotating advertisements that appear in the Program Guide. |
| CAB | A .cab file. |

| | |
|---|---|
| CDF_CHANNEL | An Internet channel. The client unpackages the files for this channel in the Web browser cache. |
| CDF_SW_DST_CHANNEL | This type is reserved for future use. |
| INVALID | Invalid content. |

## Gatherer

[This is preliminary documentation and subject to change.]

The Gatherer, Crawler.exe, is a component that runs at the command prompt and gathers a *channel* from the Web and passes it to the Cache Server. The Gatherer collects all the files specified in the *Channel Definition Format* (CDF) file defining the Internet channel.

If the Gatherer is unsuccessful at gathering a channel, it attempts to regather the channel five minutes later.

The Gatherer supports three modes of operation, allowing the following network environments:

- The Gatherer is installed on a computer directly connected to the Internet, with a correctly operating Domain Name System (DNS). In this mode, the Gatherer has full functionality with Hypertext Transport Protocol (HTTP) and File Transfer Protocol (FTP). No additional configuration is required.
- The Gatherer is installed on a local area network (LAN) using the Microsoft® Proxy Server with the Windows Sockets (WinSock) Proxy. If the WinSock Proxy is set up to pass HTTP and FTP requests and to provide name service, the Gatherer has full functionality with HTTP and FTP. No additional configuration is required.
- The Gatherer is installed on a LAN behind a firewall without WinSock Proxy, using a standard HTTP proxy server compatible with the *CERN* standard. In this mode, the Gatherer is not able to gather any FTP sites or HTTP sites whose server is set to redirect requests to Internet Protocol (IP) addresses. In this case, you must configure the system options to use an HTTP proxy. You can do this using the Internet Channel Broadcast Server Manager.

The Gatherer does not gather files for a channel that is currently being packaged. If the Packager is packaging a channel that the Gatherer is scheduled to gather, the Gatherer waits until the Packager has finished packaging the channel files. This functionality ensures that all packaged files originate from the same version of the gathered channel.

## Cache Server

[This is preliminary documentation and subject to change.]

The Cache Server, Wc_serv.exe, is a component that runs at the command prompt and manages the files collected from the Web by the Gatherer. The Cache Server communicates with the Scheduler, Gatherer, and Packager to coordinate the collection, storage, and packaging of file *channels*. The Cache Server also indexes the files and stores auxiliary information about the gathered items. It stores the files in the cache directory. This directory is specified during the Internet Channel Broadcast server installation. The cache directory location is stored in the **cache_dir** registry entry, discussed under System Options for Internet Channel Broadcasting.

In the cache directory, cached files are organized by channel into subdirectories. For example, if the server is installed and the cache directory location remains the default location, the files for the Microsoft® *Sidewalk*™ channel are stored in C:\Program Files\Webcast\Webcache\www.sidewalk.com.

Only one instance of the Cache Server should run on any specified computer. If you start another instance, the second instance stops immediately.

Cache Server supports commands typed at the command prompt. For more information, see Cache Server Commands for Command-Line Use.

## Packager

[This is preliminary documentation and subject to change.]

The Packager, Xmitter.exe, is a component that runs at the command prompt and packages files for efficient transmission. After the *channel* files are gathered and stored in the cache directory, the Packager automatically packages the files and forwards the packaged files to the package directory.

Packaging formats the files into a format suitable for efficient *File Transfer Service* (FTS) transmission. Packager adds header information that specifies information about the packaged files, such as the Internet channel the files were gathered for and the date and time they were gathered. The most efficient package size for the Internet channel broadcasting system is 1 megabyte. The Packager creates packages as close to 1 megabyte in size as possible while maintaining the channel's file structure.

Each time the Packager is run for a channel, it creates a unique directory within the destination directory, usually within the Store-and-Forward Server directory. The Packager writes the package files to this directory, naming them sequentially, for example 000.pkg, 001.pkg, and so on.

The Packager does not package files for a channel that is currently being gathered. If the Gatherer starts gathering a channel that the Packager is packaging, the Packager stops and wait until the Gatherer has updated the channel files. This functionality ensures that packaged files originate from the same version of the channel and contain the latest content.

## Store-and-Forward Server

[This is preliminary documentation and subject to change.]

The Store-and-Forward (SAF) Server, Saf.exe, monitors the directory where the Packager places files. The SAF Server sends announcements about these files using *Session Announcement Protocol*/*Session Description Protocol* (SAP/SDP). The server also sends the files themselves, using the *File Transfer Service* (FTS) protocol to transmit the files as *multicast* packets on a LAN or to a *Microsoft Multicast Router* (MMR) for the head end. In the latter case, the head end then broadcasts the Internet *channel* content to users.

The SAF Server constantly sends files. When all files in the directory have been sent, the SAF Server resends them again, starting with the first file in the directory. This round-robin mechanism ensures that updated files are continuously broadcast to the clients.

The SAF server uses a pool of IP addresses and ports to transmit packaged files. This enables the filter on the client to validate the package and prevents the client from locking up in the event that a package is dropped.

For further information on the SAF Server, see the following topics:

- SAF Server Organization
- IP Address Setup for SAF Server
- SAF Server and File Management
- SAF Server Modes

### SAF Server Organization

[This is preliminary documentation and subject to change.]

Packaged files in the SAF Server directory are organized in subdirectories named for the channels being gathered. For example, if the packaged file directory is C:\Program Files\Webcast\Pkgs\, the *Sidewalk* channel packaged files are stored in a directory such as "C:\Program Files\Webcast\Pkgs\Group-Sidewalk-19970924185304," where the numbers after the group name specify the date that the files were packaged.

The Internet Channel Broadcast server architecture separates the packaging and forwarding functions into the Packager and the SAF Server. This functionality means you can remotely administer the Internet Channel Broadcast server and yet maintain reliable transmissions of packaged Internet channel broadcasting files.

For example, if it is inconvenient to maintain the Internet Channel Broadcast server at the broadcast *head end*, you can install all of the server components except the SAF Server on a remote computer.

The SAF Server you might then install at the head end, or on a computer with a reliable connection to the head end. In this case, if the connection between the main server and the SAF Server fails the SAF Server can continue to rebroadcast files in the SAF Server directory until the network connection to the server is reinstated. This organization ensures continuous broadcasts of Internet channel broadcasting files.

If the SAF Server and Packager were not separated, you would either have to administer the server at the head end to maintain reliable transmissions of files, or accept the fact that your Internet channel broadcasting transmissions would stop whenever the connection to the head end was lost.

Installing the Packager and the SAF Server on separate servers also helps distribute the computing and storage loads.

## IP Address Setup for SAF Server

[This is preliminary documentation and subject to change.]

The SAF Server uses several *Internet Protocol* (IP) multicast address and port pairs, one for sending announcements about forthcoming files and a pool of IP addresses and ports for sending the actual files. The client constantly monitors the announcement address to determine whether to accept a particular file arriving at the file address. When the client receives an announcement about a file that it wishes to receive it uses the *File Transfer Service* (FTS) receiver, Nsfts.dll, in the Microsoft® NetShow® server to receive the packaged files.

The SAF server transmits packages using a rotating pool of IP addresses. This enables the client to better validate the incoming packages, and to timeout should a package fail to arrive. For example, if the SAF server is configured to use a pool of two IP addresses, and the file IP address and port are configured to 233.17.43.1 and 1781, the first package will be transmitted on 233.17.43.1/1781, the second package on 233.17.43.2/1782, the third package on 233.17.43.1/1781, and so on.

You must be careful when setting up IP addresses and ports for the SAF server. The Internet Channel Broadcast client cannot run on a LAN where more than one SAF Server uses the same file IP address or port option. Both the IP address and port option must be unique for each SAF Server.

In addition, the announcement IP address must be unique on the LAN, because clients can potentially receive multicast packets from multiple sources. A unique IP address ensures that the client can receive all the packages of a transmission together. If the IP address is not unique and two sources transmit packages on the same IP address, the client might receive two packages named 001.pkg and be unable to distinguish the two sets of packaged files.

## SAF Server and File Management

[This is preliminary documentation and subject to change.]

If you run the Packager but not the SAF Server, the storage medium that holds the SAF Server directory or other destination directory for packaged files fills up. The SAF Server deletes a packaged file when it determines that a different file has a newer version of the same information. If the SAF Server is not running, files with older information are not deleted.

If you discontinue gathering and transmitting a channel, you should either enable a Packager garbage collector or manually delete the packaged files for that channel. To enable a Packager garbage collector, set the registry value **gc_period** to nonzero. For more information on how to do so, see Configuring System Options and System Options for Internet Channel Broadcasting.

**Note** If you set **gc_period** to a nonzero value, you must stop and restart the Packager to make the change take effect.

### SAF Server Modes

[This is preliminary documentation and subject to change.]

The SAF Server can run in two different modes. It either multicasts directly to a local Ethernet (the default mode), or it makes a *Transmission Control Protocol/Internet Protocol* (TCP/IP) connection to an MMR that forwards the packets to a service provider's broadcast system.

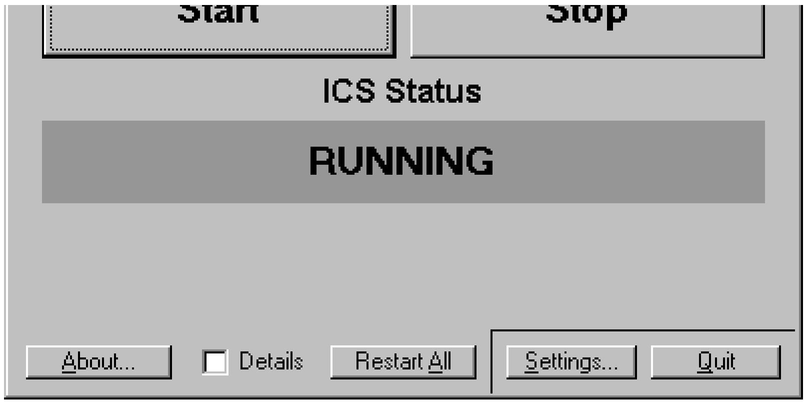
For more information, see Starting and Stopping the Server.

## Internet Channel Broadcast Server Manager

[This is preliminary documentation and subject to change.]

Using the Internet Channel Broadcast Server Manager, shown in the following illustration, you can start and stop the Internet Channel Broadcast server and change the settings for system options for Internet channel broadcasting.

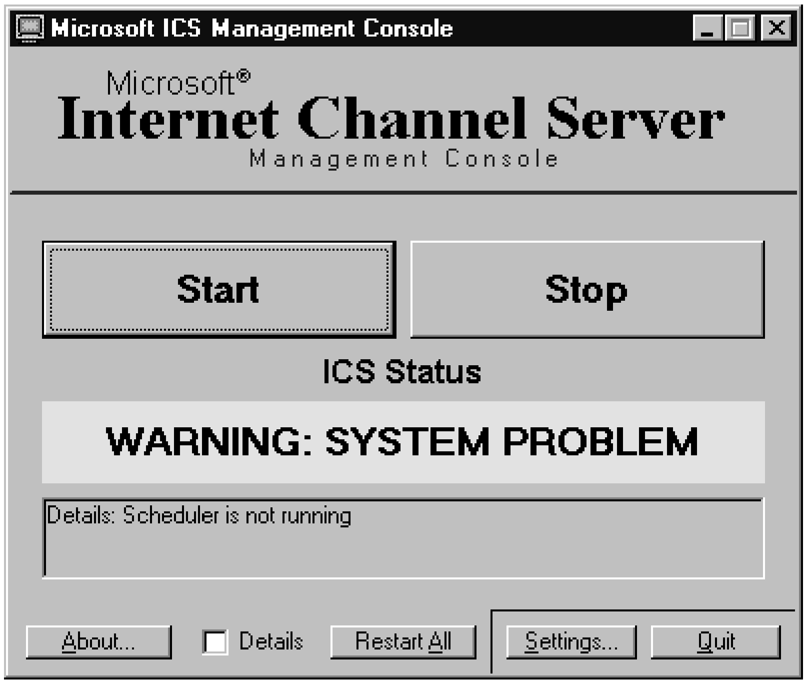
The following dialog box appears when you start the server manager. You can use the **Start** and **Stop** buttons to start and stop the entire Internet Channel Broadcast server as a unit.



ICS Status

RUNNING

The **ICS Status** box indicates the status of the server. When the server is stopped, the box is red. When the server is running without errors, the box is green. When the server encounters an error, the box turns yellow and displays a short message describing the problem, as shown in the following illustration.

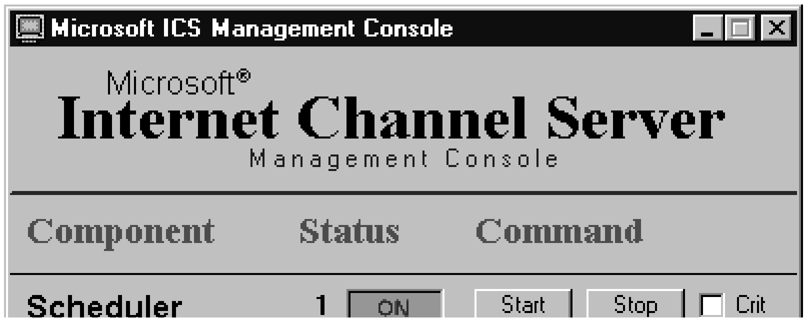


ICS Status

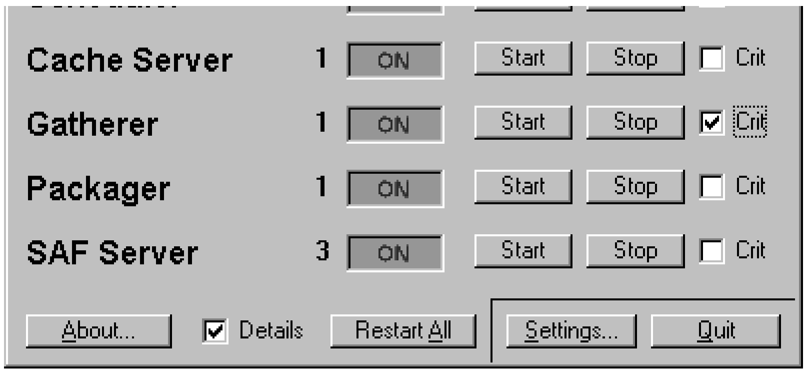
WARNING: SYSTEM PROBLEM

Details: Scheduler is not running

If you need to administer the Internet Channel Broadcast server components individually, for example if you installed the SAF Server on a separate computer than the rest of the server components, click the **Details** check box to select it. Doing so displays the dialog box shown following. Using this dialog box, you can start and stop the server components individually.



| Component | Status | | Command | | |
|---|---|---|---|---|---|
| Scheduler | 1 | ON | Start | Stop | Crit |


Cache Server 1 ON Start Stop Crit
Gatherer 1 ON Start Stop Crit
Packager 1 ON Start Stop Crit
SAF Server 3 ON Start Stop Crit
About...
Details
Restart All
Settings...
Quit

When a server component starts, a number appears in the **Internet Channel Broadcast Server Manager** dialog box to the left of the button under **Status**. This number indicates how many instances of the component are currently running.

For more information on using the server manager, see the following topics:

- Starting and Stopping the Server
- Starting and Stopping Server Components
- Configuring System Options
- Specifying Critical Components
- Restarting Server Components

## System Options for Internet Channel Broadcasting

[This is preliminary documentation and subject to change.]

The following system options can be set by clicking the **Settings** button in the Internet Channel Broadcast Server Manager and changing option settings in the dialog box that appears. If you do not set these values, the server uses the default values set during the server installation.

The values you set for these options are stored in the registry, in subkeys of **HKLM\Software\Microsoft\Webcast\**. Therefore, if you change an option setting you must stop and restart the corresponding server component before the changes take effect. For more information on stopping and restarting the server, see Starting and Stopping the Server.

The following table lists and describes options you can set on the **General** tab. These options affect the server as a whole. The values you set are stored in the **\General** subkey in the registry.

| Option | Registry key | Description |
|---|---|---|
| **ICS Path** | **bin_dir** (String) | The directory that contains the server executable files. The default is C:\Program Files\Webcast\Bin. |

| | | |
|---|---|---|
| **Logging Directory** | **logdir** (String) | The directory where the Internet channel broadcasting component writes log files containing status and error messages. The default directory is C:\Program Files\Webcast\Logs. |
| **Cache Directory** | **cache_dir** (String) | The Cache Server directory in which the server stores gathered files prior to packaging. The default is C:\Program Files\Webcast\Webcache. |
| **Packing Directory** | **pkg_dir** (String) | The Packager directory, where packaged files are stored to be picked up by the Store-and-Forward Server. To specify a remote computer, enter a universal naming convention (UNC) path, for example \\*HeadEnd**PackageDir*.The default is C:\Program Files\Webcast\Pkgs. |
| **Database Directory** | **webcast_dir** (String) | The directory that contains the Scheduler database. The default is C:\Program Files\Webcast. |

The following table lists and describes options you can set on the **Connection** tab. The values you set are stored in the **\crawler** and **\saf** subkeys in the registry.

| **Option** | **Registry key** | **Description** |
|---|---|---|
| **HTTP Proxy** | **proxy** (String) | The Hypertext Transport Protocol (HTTP) proxy server to use. If you do not specify a proxy here, the Gatherer works as if a direct connection exists to the Internet. |
| **Bridge Server Name** | **bdnserver** (String) | The name of the Microsoft Multicast Router (MMR) to use. If you do not specify an MMR here, the server does not tunnel through an MMR. |
| **Announce Address** | **annaddr** (String) | The *Internet Protocol* (IP) *multicast* address to which announcements are sent. This value is initially set to 233.17.43.1. If the default value conflicts with an address on the local network, specify a nonconflicting address for this option and also specify a nonconflicting address for the client in the Announcement Listener registry settings. |

| | | |
|---|---|---|
| **Announce Port** | **annport** (**DWORD**) | A value indicating the *multicast* port to which announcements are sent. The port value is initially set to 1780. If the default value conflicts with the value of a port in use on the local network, specify a nonconflicting port for this option and also specify a nonconflicting address for the client in the Announcement Listener registry settings. |
| **Max. Throughput** | **kbps** (**DWORD**) | The maximum kilobits of data per second for multicast. The default rate is 100 kilobits per second (Kbps) over a LAN and 800 Kbps when tunneling to an MMR. |
| **Maximum Connects** | **maxconns** (**DWORD**) | The maximum number of simultaneous connections the Gatherer can make to a single Web host. Typically, unless a service provider has made special arrangements with the Web server administrator, this option should be set to 1. |
| **Local Address** | **localaddr** (String) | The IP address of the network interface the SAF Server uses send multicast packets. The local address enables the SAF Server to run on multihomed computers, in other words computers that have more than one network interface and address. This value is ignored when tunneling through an MMR. |
| **Multicast Address** | **fileaddr** (String) | The IP multicast address to which data is sent. This value is initially set to 233.17.43.2. If the default value conflicts with an address on the local network, specify a nonconflicting address for this option. |
| **Multicast Port** | **fileport** (**DWORD**) | A value indicating the multicast port to which data is sent. The port value is initially set to 1781. If the default value conflicts with the value of a port in use on the local network, specify a nonconflicting port value for this option. |

| | | |
|---|---|---|
| **Time-To-Live** | **ttl** (**DWORD**) | The time-to-live field for multicast packets. The default value is 1; to multicast through routers over a LAN, enter a higher value. The time to live can be any of the following values:<br><br>0 Restricted to same host<br><br>1 Restricted to same subnetwork<br><br>32 Restricted to same site<br><br>64 Restricted to same region<br><br>128 Restricted to same continent<br><br>256 Unrestricted in scope |
| **GC Period** | **gc_period** (**DWORD**) | The interval between garbage collection attempts in seconds. Garbage collection in this context means checking the Scheduler database and removing items from the package store that no longer appear in the schedule. By default this number is zero, meaning that garbage collection is not performed. |

The following table lists and describes options you can set on the **Advanced** tab. The values you set are stored in the **\xmitter** and **\wc_serv** registry subkeys.

| **Option** | **Registry key** | **Description** |
|---|---|---|
| **Pkg Buffer Height** | **buffer_height** (**DWORD**) | The number of lines in the buffer for the Packager's Command Prompt window. If this value is greater than the value for the **Pkg Window Height** option, the window appears with a scroll bar. The **Pkg Buffer Height** value must be greater than or equal to the **Pkg Window Height** value. The default value is 50 lines. |
| **Pkg Window Height** | **window_height** (**DWORD**) | The number of lines displayed in the Packager's Command Prompt window. The default value is 50 lines. |

| | | |
|---|---|---|
| **Pkg Window Width** | **window_width** (**DWORD**) | The characters per line in the Packager's Command Prompt window. The default value is 110 characters per line. |
| **Pkg Error Level** | **window_height** (**DWORD**) | The minimum amount of free space, in megabytes, required on the disk that stores the packaged files. If the amount of free space drops below this value, the Packager displays an error message to the user. The default is 1 megabyte. |
| **Pkg Warn Level** | **pkg_warn_level** (**DWORD**) | The recommended amount of free space, in megabytes, on the disk that stores the packaged files. If the amount of free space drops below this value, the Packager displays a warning. The default is 2 megabyte. |
| **Cache Static Lines** | **cache_static_ lines** (**DWORD**) | The number of lines reserved at the top of the Cache Server's Command Prompt window to display connections to other Internet channel broadcasting components, such as the Packager. |
| **Cache Buffer Height** | **buffer_height** (**DWORD**) | The number of lines in the buffer for the Cache Server's Command Prompt window. If this value is greater than the value for the **Cache Window Height** option, the window appears with a scroll bar. The **Cache Buffer Height** value must be greater than or equal to the **Cache Window Height** value. The default value is 50 lines. |
| **Cache Window Height** | **window_height** (**DWORD**) | The number of lines displayed in the Cache Server's Command Prompt window. The default value is 50 lines. |
| **Cache Window Width** | **window_width** (**DWORD**) | The characters per line in the Cache Server's Command Prompt window. The default value is 110 characters per line. |
| **Temp Error Level** | **temp_error_ level** (**DWORD**) | The minimum amount of free space, in megabytes, required on the disk that stores the temporary files generated by the Packager during the file-packing process. The path to this directory is retrieved using the Microsoft® Win32® application |

| | | |
|---|---|---|
| | | programming interface (API) function **GetTempPath**, which reads the environment variables. If the amount of free space drops below this value, the Packager displays an error message. The default value is 1 megabyte. |
| **Temp Warn Level** | **temp_warn_ level** (**DWORD**) | The recommended amount of free space, in megabytes, on the disk that stores the temporary files generated by the Packager during the file-packing process. The path to this directory is retrieved using **GetTempPath**, which reads the environment variables. If the amount of free space drops below this value, the Packager displays a warning. The default value is 2 megabytes. |

The following table lists server options you cannot set using Webcast Server Manager. These values can only be changed by editing the registry values directly. They are stored in the **\saf** subkey.

| **Option** | **Registry key** | **Description** |
|---|---|---|
| N/A | **delay_time** | The number of milliseconds between the time that the SAF server an announcement and the time that it sends the packaged files. |
| N/A | **ippoolsize** | The number of IP address and port pairs to allocate for file transmission. The SAF server increments the values of **fileaddr** and **fileport** to rotate through the pool of IP addresses.<br><br>For example, if the SAF server is configured to use a pool of two IP addresses, and **fileaddr** is 233.17.43.1 and **fileport** is 1781, the SAF server transmits the first package on 233.17.43.1/1781, the second package on 233.17.43.2/1782, the third package on 233.17.43.1/1781, and so on. |
| N/A | **session_time** | The number of seconds in the SDP session time. In other words, the announcement will time out after **session_time** seconds. |

The following table lists server options you cannot set using Webcast Server Manager. These values control the logging behavior of the server. They can only be changed by editing the registry values directly.

| **Option** | **Registry key** | **Description** |
|---|---|---|
| N/A | **loglev** | This value controls the amount of detail written to the log files. A setting of 1 indicates normal log details, whereas a setting of 3 indicates a higher level of detail. Note that setting the logging level to a value such as 5 can quickly fill up your computer's storage.<br><br>This value is set under the subkey corresponding to the component for which you are setting the verbosity. For example, to set the SAF server verbosity, add a loglev value to the **\saf** subkey. |
| N/A | **logsize** | The maximum size of log files, in bytes. This value is set under the **\General** subkey. When a server component exceeds this limit it closes the current log file and creates a new one. The server only keeps the three most current log files for any component. Older log files are deleted.<br><br>The default value for logsize is 2048000, or 2 MB. The minimum size of log files is 1MB. |

**Note** You can override existing option settings by specifying arguments when the Internet Channel Broadcast server is started from the command line. This approach is typically used only when testing the Internet Channel Broadcast server components. For more information, see Command-Line Initialization for the Internet Channel Broadcast Server.

To locate documentation on **GetTempPath**, see Further Server Information.

# Using Internet Channel Broadcast Server

[This is preliminary documentation and subject to change.]

The Internet Channel Broadcast server is a set of components administered by a *service provider*. The server gathers files from the Web, packages them for transmission, and then transmits them using a broadcast medium, for example as a signal coded into the *vertical blanking interval* (VBI) of an analog television broadcast or as packets over a local area network (LAN).

For information about how to use the Internet Channel Broadcast server, see the following topics:

- Installing the Server
- Configuring System Options
- Starting and Stopping the Server
- Starting and Stopping Server Components
- Restarting Server Components
- Specifying Critical Components
- Specifying Channels to Broadcast
- Logging Server Events

## Installing the Server

[This is preliminary documentation and subject to change.]

The Internet Channel Broadcast server components require the Microsoft® Windows NT® operating system, version 4.0 or higher.

**To install the Internet Channel Broadcast server software**

1. Run the installation program, Webcast.exe. You can do this either from the command prompt, or by double-clicking the icon in Windows NT Explorer.

   A dialog box appears that asks if you want to install the Internet Channel Broadcast server.

2. Click **Yes**.

   A second dialog box appears that asks in which directory the server software should be installed.

3. Enter a directory, or use the default value displayed in the text box, C:\Program Files\Webcast, and click **OK**.

   If the directory that you specified does not already exist, a dialog box appears asking you if you wish to create the directory. Click **Yes**.

4. The installation program installs the server files. A dialog box appears asking if you want to restart your computer.
5. Click **Yes**.

When the computer restarts, the server installation is complete.

The Scheduler database is installed in the directory that you specified in step 3 preceding. During installation, the following subdirectories are created under the directory where you installed the Internet Channel Broadcast server:

- \Bin, which contains the binary files for the Internet Channel Broadcast server.
- \Pkgs, which is where the Internet Channel Broadcast server stores packaged files.
- \Webcache, which is where the Internet Channel Broadcast server stores files prior to packaging them for broadcast.
- \Logs, which is where the components of the Internet Channel Broadcast server write status and error messages.

Both the Cache Server directory (by default, C:\Program Files\Webcast\Webcache) and the Packager directory (by default, C:\Program Files\Webcast\Pkgs) must have sufficient storage space to contain all of the Web sites specified in the Scheduler database. The amount of space required depends on the number and size of the files the Internet Channel Broadcast server is gathering and transmitting. The Cache Server directory may require more space than expected because it stores many small files, especially on a partition formatted in 16-bit file allocation table (FAT) format.

For these reasons, you may prefer to install the Store-and-Forward (SAF) Server and the Packager on different computers.

**To install the SAF Server and Packager on different computers**

1. Run the installation program on both computers, as described preceding in "To install the Internet Channel Broadcast server software."
2. Start the Gatherer, Cache Server, Packager, and Scheduler on the first computer.
3. Start only the SAF Server on the second computer.
4. On the first computer, specify the directory to hold packaged files in the system settings using universal naming convention (UNC) format, for example:

   **\\Saf_machine\Share_dir\Webcast\Pkgs**

   The Internet Channel Broadcast server is now configured to forward the packaged files to the SAF Server computer.

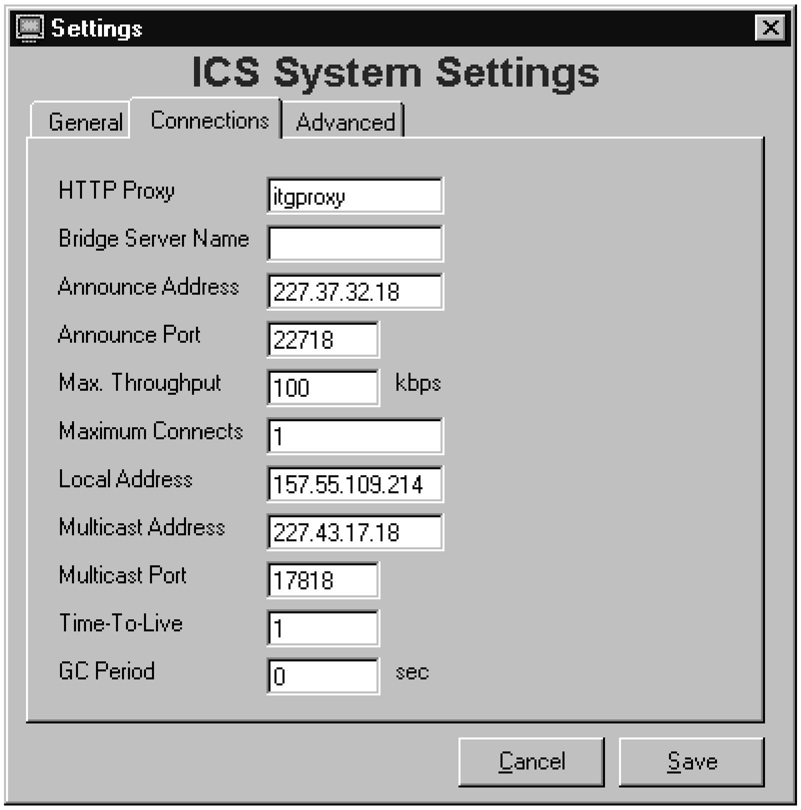
5. If you used Internet Channel Broadcast Server Manager to start the components, close the server manager by clicking **Quit**. If you started the components from the command line, no further action is necessary.

## Configuring System Options

[This is preliminary documentation and subject to change.]

The current settings for system options are stored in the registry. You can use the Internet Channel

Broadcast Server Manager to change these settings, as shown in the following illustration.



HTTP Proxy: itgproxy
Bridge Server Name:
Announce Address: 227.37.32.18
Announce Port: 22718
Max. Throughput: 100 kbps
Maximum Connects: 1
Local Address: 157.55.109.214
Multicast Address: 227.43.17.18
Multicast Port: 17818
Time-To-Live: 1
GC Period: 0 sec

**To configure the Internet Channel Broadcasting system options**

1. Start the Internet Channel Broadcast Server Manager, either by running Serv_mgr.exe from the command line, or by double-clicking the Server Manager icon in Windows NT Explorer.
2. Click the **Settings** button. The **ICS System Settings** dialog box appears, as shown in the preceding illustration.
3. Click the tab that corresponds to the type of option you want to set, either **General**, **Connections**, or **Advanced**.
4. Edit the option settings. For more information on specific option settings, see Scheduler Database Schema.
5. Click **Save**. The server manager updates the Internet Channel Broadcasting registry entries with the new values.
6. Stop and restart the components affected by the changes by clicking the **Restart All** button. Restarting the components is necessary for the new settings to take effect. For more information, see Starting and Stopping the Server.
7. Click **Quit** to close the server manager.

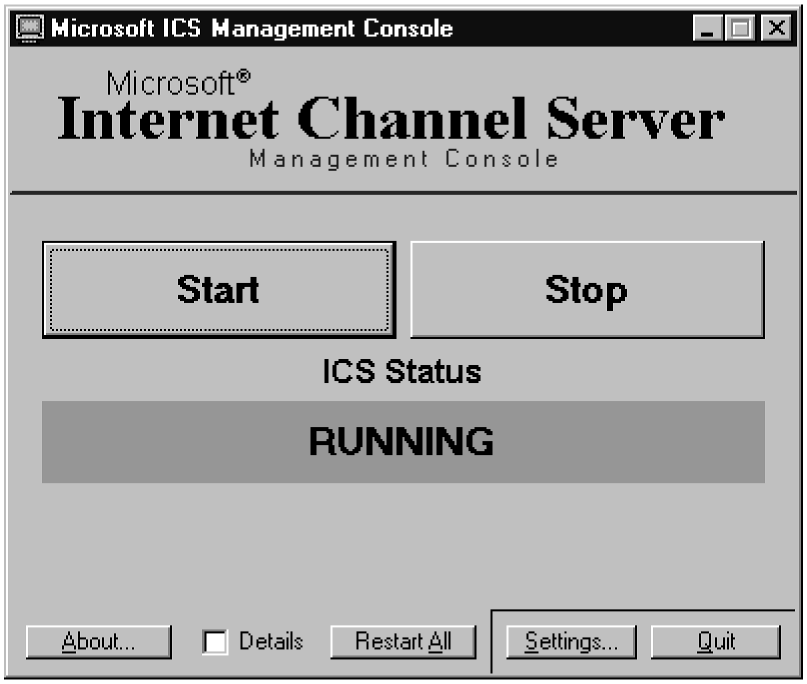
For more information about the configuration options, see Scheduler Database Schema.

## Starting and Stopping the Server

[This is preliminary documentation and subject to change.]

Before you start the server, you must create a list of the *channels* you plan to broadcast in the Scheduler database, Groupdb.mdb. You can add, delete, or modify these entries using Microsoft® Access. For more information about the information contained in the Scheduler database, see Scheduler Database Schema.

Once you have configured the Scheduler database, you can start the server components. Typically, you start and stop the server components by using the Internet Channel Broadcast Server Manager, shown in the following illustration.



**To start the Internet Channel Broadcast server**

1. Click **Start**.
2. If you do not need to further administer the server, click **Quit** to close the server manager.

**To stop the Internet Channel Broadcast server**

1. Click **Stop**.
2. Click **Quit** to close the server manager.

For information on other options for starting and stopping the Internet Channel Broadcast server, see the following topics:

- Starting and Stopping Server Components
- Command-Line Initialization for the Internet Channel Broadcast Server

## Starting and Stopping Server Components

[This is preliminary documentation and subject to change.]

In addition to starting and stopping the entire Internet Channel Broadcast server as a unit, you can start and stop the individual server components. Doing so is useful, for example, if you installed the SAF Server on a separate computer than the rest of the server components

You can start individual components either from the Internet Channel Broadcast Server Manager or the command line. For information about starting the server components from the command line, see Command-Line Initialization for the Internet Channel Broadcast Server.

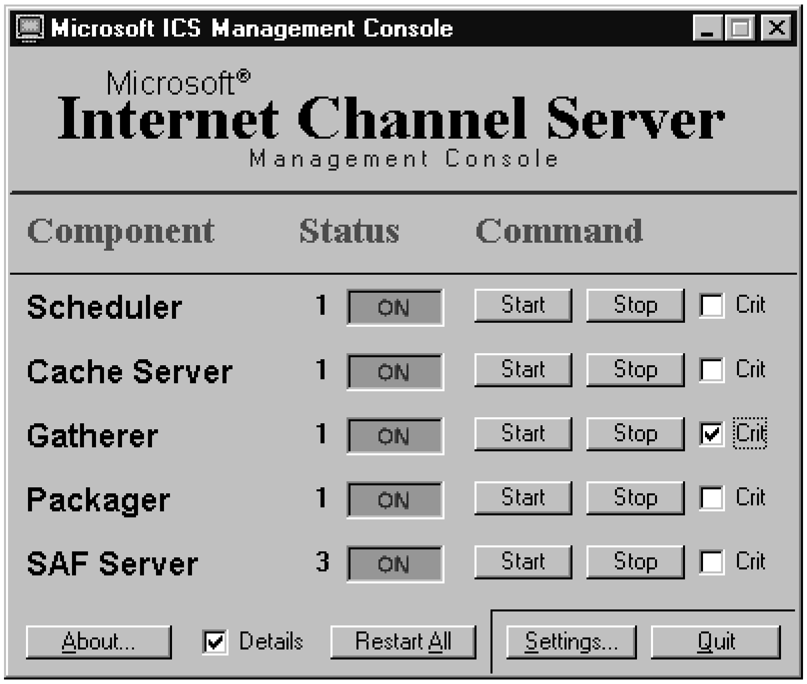
You must start the server components in the following order:

1. Scheduler
2. Cache Server
3. Gatherer
4. Packager
5. Store-and-Forward Server

**To start a server component**

1. In the Internet Channel Broadcast Server Manager dialog box, check the **Details** check box.

   The **Details** dialog box appears, as shown in the following illustration.



| Component | Status | | Command | | |
|---|---|---|---|---|---|
| Scheduler | 1 | ON | Start | Stop | ☐ Crit |
| Cache Server | 1 | ON | Start | Stop | ☐ Crit |
| Gatherer | 1 | ON | Start | Stop | ☑ Crit |
| Packager | 1 | ON | Start | Stop | ☐ Crit |
| SAF Server | 3 | ON | Start | Stop | ☐ Crit |

2. Click the **Start** button to the right of the component's name.

3. If you do not need to further administer the server, click **Quit** to close the server manager.

**To stop a server component**

1. From the **Details** dialog box, click the **Stop** button to the right of the component's name.
2. Click **Quit** to close the server manager.

Remote administration is not currently implemented in the Internet Channel Broadcast Server Manager. Thus, if server components are installed on multiple computers, the configuration and administration possible with each instance of the server manager varies depending on the components installed on the computer the instance resides on.

For example, if you install the Scheduler, Cache Server, Gatherer, and Packager on one computer and the SAF Server on another, you can start the first four components using the server manager running on the first computer. However, you start the SAF Server using the server manager running on the second computer.

## Restarting Server Components

[This is preliminary documentation and subject to change.]

You can use the Internet Channel Broadcast Server Manager to restart one or more of the server components at a time. Restarting the components is necessary, for example, when you have changed the system configuration options. The new settings do not take effect until the affected component is restarted.

**To restart a single server component**

1. Click to select the **Details** check box in the server manager. The **Details** dialog box appears.
2. Click **Stop** to the right of the component that you want to restart.
3. Click **Start** to the right of that component.

**To restart all of the server components**

- Click **Restart All**.

If multiple instances of a component are running when you click **Restart All**, all instances are shut down and then restarted. For example, if two instances of the Gatherer are running when you click **Restart All**, both instances shut down and the server manager creates two new instances.

**Note** The components cannot send output to the command prompt when running in select mode. In *select mode*, you can highlight text for cut-and-paste operations. For example, suppose you right-click the icon for the Microsoft® MS-DOS® operating system at the top left of a Command Prompt window. You then click **Edit** and then **Mark**. The window is now set to select mode.

The server manager monitors the mode of the Command Prompt windows that the Internet channel broadcasting components run in. To alert you that a window is in select mode and therefore blocked from displaying output, the server manager causes the title bar of the Command Prompt window to flash.

## Specifying Critical Components

[This is preliminary documentation and subject to change.]

You can specify a Internet Channel Broadcast server component as critical. The server manager monitors all instances of critical components and automatically restarts any instances that shut down in atypical fashion. Typical shutdown, such as when you click the **Stop** button, does not cause the server manager to automatically restart the component.

**To specify a server component as critical**

- In the **Details** dialog box of Internet Channel Broadcast Server Manager, click to select the **Crit** check box associated with the component you want to specify as critical.

The component remains a critical component until you specify it as noncritical or close the server manager.

**To specify a server component as noncritical**

- In the **Details** dialog box, click to clear the **Crit** check box associated with the component you want to specify as noncritical.

The component is no longer a critical component.

## Specifying Channels to Broadcast

[This is preliminary documentation and subject to change.]

Before the Internet Channel Broadcast server can start collecting and broadcasting files, you must create a list of *channels* to be broadcast in the Scheduler database and set options that describe how each channel is collected and broadcast.

You can use Microsoft Access to administer the database file, Groupdb.mdb. To configure a new channel for broadcasting, simply open the Groups table in Groupdb.mdb and add a record. Specify values for the fields as described in the topic Scheduler Database Schema. When you are finished, close the database.

Note that the service provider, the person or organization administering the Internet Channel Broadcast server, does not usually create the Internet channels that are broadcast. The server can collect and broadcast any Web site that has created a *Channel Definition Format* (CDF) file and published it on the Web. This functionality is analogous to television broadcasting, in which a local television station broadcasts content that it has not created.

## Logging Server Events

[This is preliminary documentation and subject to change.]

Each component of the Internet Channel Broadcast server logs data about its status and actions to a text file in the log directory. The location of this directory is specified in the HKLM\Softwar\Microsoft\Webcast\General\**logdir** registry value.

You can set the amount of detail logged by a component and the maximum log file size for the server. To do this configure the **loglev** and **logsize** regitry values. For more information, see System Options for Internet Channel Broadcasting.

When a server component's log file reaches the maximum size, the component closes the current log file and opens a new one. The server keeps only the three most recent log files for any components. Older log files are deleted.

# Internet Channel Broadcast Server Reference

[This is preliminary documentation and subject to change.]

The following topics provide reference information on specific Internet Channel Broadcast server components:

- Cache Server Commands for Command-Line Use
- Command-Line Initialization for the Internet Channel Broadcast Server
- Command-Line Initialization for the Scheduler
- Command-Line Initialization for the Gatherer
- Command-Line Initialization for the Cache Server
- Command-Line Initialization for the Packager
- Command-Line Initialization for the Store-and-Forward Server
- Internet Channel Broadcast Announcement Format

## Cache Server Commands for Command-Line Use

[This is preliminary documentation and subject to change.]

When Cache Server is running, it supports entry of commands at its command prompt. The following table lists and describes these commands. In the following descriptions, *gspec* is a channel specification following the rules for the Spec field described in Group Specification Format.

| Command | Description |
|---|---|
| **debug** | Lists the location where the Cache Server is logging status and error messages. |
| **debuglev** *N* | Lists the current verbosity setting for the debugging display, or sets the verbosity to the value specified by *N*. Zero specifies no display; 1 is usual verbosity; values up to 6 are increasingly verbose. If *N* is higher than the current value for the **loglev** command, the **loglev** value is changed to match *N*. If *N* is not specified, the debug level is set to 1. |
| **exit** | Closes the Cache Server. |
| **groups** | Lists the names of the channels currently present in the Internet channel broadcasting cache. |
| **help** | Lists the available commands. |
| **list** *gspec* | Lists the files currently in the cache for the channel specified by *gspec*. |
| **loglev** *N* | Lists the current verbosity setting for information sent to the log file, or sets the verbosity to the value specified by *N*. Zero specifies no logging; 1 is default verbosity; values up to 6 are increasingly verbose. For example, a verbosity of 1 logs a message when you set the logging directory. In contrast, a verbosity of 6 logs information about each step in the process of setting the directory, including the status of individual threads. |
| **objinfo** *url* | Lists information about the URL specified by *url*. This URL must be fully qualified. |
| **quiet** | Sets the debugging display verbosity to level 1. |
| **quit** | Closes the Cache Server. |
| **remove** *gspec* | Deletes all cache files for the channel specified by *gspec*. |
| **sh** | Sets the debugging display verbosity to level 1. |

| | |
|---|---|
| **sitemap** | Lists the names of the sites currently present in the cache. Each site is paired with the name of its associated Internet channel. |
| **sites** | Lists the names of the sites currently present in the cache, for example www.microsoft.com. |
| **size** *gspec* | Lists the size, in bytes, of all files in the channel specified by *gspec*. |
| **stats** *gspec* | Displays the cached files matching the *gspec* specification, and their sizes. |

## Command-Line Initialization for the Internet Channel Broadcast Server

[This is preliminary documentation and subject to change.]

Typically, you use the Internet Channel Broadcast Server Manager to start and stop the various components of the Internet Channel Broadcast server. However, you can also start these components from the command line.

To do so, you must first enter records for all Internet channels in the Scheduler database. Then, from the directory that contains the Internet channel broadcasting components, start each component in the following order using the following commands:

- **start wbcsched**, to start the Scheduler
- **start wc_serv**, to start the Cache Server
- **start crawler**, to start the Gatherer
- **start xmitter**, to start the Packager
- **start saf**, to start the Store-and-Forward Server

For more information, see the following topics:

- Command-Line Initialization for the Scheduler
- Command-Line Initialization for the Cache Server
- Command-Line Initialization for the Gatherer
- Command-Line Initialization for the Packager
- Command-Line Initialization for the Store-and-Forward Server

## Command-Line Initialization for the Scheduler

[This is preliminary documentation and subject to change.]

Typically, you start the Scheduler by using the Internet Channel Broadcast Server Manager. However, you can also start the Scheduler from the command line. You typically do so when testing the Internet Channel Broadcast server.

To start the Scheduler from the command line, use the following syntax:

**wbcsched**

## Command-Line Initialization for the Gatherer

[This is preliminary documentation and subject to change.]

Typically, you start the Gatherer by using the Internet Channel Broadcast Server Manager. When the Gatherer is started this way, it connects to the Scheduler to find out what channels need to be collected from the Web. In this case, the Scheduler must already be running when the Gatherer is started.

However, for testing purposes you can start the Gatherer from the command line, specifying directly the files it should gather. In this case, you do not need to have previously started the Scheduler.

To start the Gatherer from the command line, use the following syntax:

**crawler** *arguments*

The following table lists and describes possible values for the optional command-line arguments specified by *arguments*.

| Argument | Description |
|---|---|
| **-maxgroups** *N* | The maximum number of channels that a single instance of the Gatherer can retrieve, as specified by *N*. When this number is reached, the Gatherer stops retrieving Web pages and creates another Gatherer instance. This new Gatherer retrieves any remaining channels. If the **-maxgroups** argument is not entered at the command line, the Gatherer uses the **maxgroups** value specified in the registry. For more information, see System Options for Internet Channel Broadcasting. |

| | |
|---|---|
| **-name** *gname gspec* | A file group, where *gname* is the channel's name as listed in the Scheduler database and *gspec* is the URL that specifies what files should be gathered for this channel. The *gspec* argument uses the same format as described in Group Specification Format. |

**Note** When you start the Gatherer from the command line, you must ensure the file channel and specification indicated by the *gname* and *gspec* arguments already exist in the Scheduler database. The Gatherer does not verify *gname* and *gspec*.

## Command-Line Initialization for the Cache Server

[This is preliminary documentation and subject to change.]

Typically, you start the Cache Server by using the Internet Channel Broadcast Server Manager. When the Cache Server is started this way, it connects to the registry to initialize its settings.

However, you can also start the Cache Server from the command line, specifying settings to use instead of those stored in the registry. You typically do so when testing the Internet Channel Broadcast server.

To start the Cache Server from the command line, use the following syntax:

**wc_serv** *arguments*

The following table lists and describes possible values for the optional command-line arguments specified by *arguments*.

| Argument | Description |
|---|---|
| **-cache_dir** *path* | Sets the location to store gathered files to the path indicated by *path*. This value supersedes the location entry in the registry. |
| **-debuglev** *N* | Lists debug information at the command prompt at the verbosity indicated by *N*. If *N* is higher than the current value for the **loglev** argument, the **loglev** value is changed to match *N*. If *N* is not specified, the debug level is set to 1. |
| **-logdir** *directory* | Sets the logging directory to *directory*. |

| | |
|---|---|
| **-loglev** *N* | Sets the level of verbosity for information sent to the log file to the level indicated by *N*. Zero specifies no logging; 1 is default verbosity; values up to 6 are increasingly verbose. For example, a verbosity of 1 logs a message when you set the logging directory. In contrast, a verbosity of 6 logs information about each step in the process of setting the directory, including the status of individual threads. |

Cache Server also supports entry of commands at its command prompt while it is running. For more information, see Cache Server Commands for Command-Line Use.

Only a single instance of the Cache Server can run on any specified computer. If you try to start another instance on the same computer, the second instance exits immediately.

## Command-Line Initialization for the Packager

[This is preliminary documentation and subject to change.]

Typically, you start the Packager by using the Internet Channel Broadcast Server Manager. When the Packager is started this way, it connects to the Scheduler to determine which channels to package. In this case, you must start the Scheduler before the Packager.

However, for testing purposes you can start the Packager from the command line, specifying directly the files it should package. In this case, you do not need to have previously started the Scheduler.

To start the Packager from the command line, use the following syntax:

**xmitter** *gname arguments*

In addition to the required argument *gname*, you can specify optional command-line arguments. The following table lists and describes the possible arguments.

| Argument | Description |
|---|---|
| *gname* | Sets the channel to package, where *gname* is the channel name as listed in the Scheduler database. |
| **-clear_saf** | Clears out the package store before new packaged items are stored there. If this argument is specified, the package store contains only items packaged by the current instance of the Packager. |

| | |
|---|---|
| **-debuglev** *N* | Lists logging information at the command prompt at the verbosity indicated by *N*. If *N* is higher than the current value for the **loglev** argument, the **loglev** value is changed to match *N*. If *N* is not specified, the debug level is set to 1. |
| **-gc_period** *N* | Specifies the interval between garbage collection attempts in seconds, as specified by *N*. Garbage collection in this context means checking the Scheduler database and removing items from the package store that no longer appear in the schedule. By default this number is zero, meaning that garbage collection is not performed. |
| **-loglev** *N* | Sets the level of verbosity for information sent to the log file to the level indicated by *N*. Zero specifies no logging; 1 is default verbosity; values up to 6 are increasingly verbose. For example, a verbosity of 1 logs a message when you set the logging directory. In contrast, a verbosity of 6 logs information about each step in the process of setting the directory, including the status of individual threads. |
| **-pkg_dir** *path* | Sets the location to store packaged files to the path indicated by *path*. This value supersedes the location entry in the registry. |
| **-pthreads** *N* | Sets the number of possible Packager threads to the value indicated by *N*. The default value is 5. Faster performance is possible if the **-pthreads** value is larger, but at higher computing overhead. |
| **-quit** | Closes the Packager after startup. The **-quit** argument is useful only with the **-clear_saf** argument. When both arguments are specified, the contents of the package store are deleted and then the Packager quits. |

## Command-Line Initialization for the Store-and-Forward Server

[This is preliminary documentation and subject to change.]

Typically, you start the Store-and-Forward (SAF) Server by using the Internet Channel Broadcast Server Manager. When the SAF Server is started this way, it connects to the registry to initialize its settings.

However, you can also start the SAF Server from the command line, specifying settings to use instead of those stored in the registry. You typically do so when testing the Internet Channel Broadcast server.

To start the SAF Server from the command line, use the following syntax:

**saf** *arguments*

The following table lists and describes possible values for *arguments*.

| **Argument** | **Description** |
|---|---|
| **-annaddr** *address* | Sends announcements to the IP address specified by *address* instead of the address listed in the registry. |
| **-annport** *port* | Sends announcements to the port specified by *port* instead of the port listed in the registry. |
| **-bdnserver** *MMRname* | Connects to the Microsoft Multicast Router (MMR) specified by *MMRname*. If this argument is not used, the SAF Server multicasts over the LAN. |
| **-debuglev** *N* | Lists logging information at the command prompt at the verbosity indicated by *N*. If *N* is higher than the current value for the **loglev** argument, the **loglev** value is changed to match *N*. If *N* is not specified, the debug level is set to 1. |
| **-fileaddr** *address* | Broadcasts files to the IP address specified by *address* instead of the address listed in the registry. |
| **-fileport** *port* | Sends files to the port specified by *port* instead of using the value set in the **Multicast Port** system option. |
| **-kbps** *N* | Sets the amount of bandwidth to use for transmitting files, in kilobits per second, to the value *N* instead of the value listed in the registry. |
| **-localaddr** *address* | Sets the IP address of the network interface the SAF Server uses to send multicast packets to the address indicated by *address*. The local address enables the SAF Server to run on multihomed computers, in other words on computers that have more than one network interface and address. This value is ignored when tunneling through an Microsoft Multicast Router (MMR). |
| **-loglev** *N* | Sets the level of verbosity for information sent to the log file to the level indicated by *N*. Zero specifies no logging; 1 is default verbosity; values up to 6 are increasingly verbose. For example, a verbosity of 1 logs a message when you set the logging directory. In contrast, a verbosity of 6 logs information about each step in the process of setting the directory, including the status of individual threads. |

| | |
|---|---|
| **-pkg_dir** *path* | Sets the location to store packaged files to the path indicated by *path* instead of the location listed in the registry. If this argument is used, the directory section of the path must be the same as that specified on the Packager command line. |
| **-ttl** *N* | Sets the time-to-live field for multicast packets to the integer *N*, instead of the value listed in the registry. *N* can be any of the following values:<br><br>0 Restricted to same host<br><br>1 Restricted to same subnetwork<br><br>32 Restricted to same site<br><br>64 Restricted to same region<br><br>128 Restricted to same continent<br><br>256 Unrestricted in scope |

## Group Specification Format

[This is preliminary documentation and subject to change.]

Previous versions of Internet channel broadcasting (formerly called webcasting in Broadcast Architecture) used the group specification format to specify which Web files should be gathered and broadcast. The current version uses Internet *channels*.

However, the Cache Server and Gatherer command-line commands still use the older group specification format. For more information on these commands, see Command-Line Initialization for the Gatherer and Cache Server Commands for Command-Line Use.

The following table describes the group specification format.

| **Group specification** | **Effect** |
|---|---|

| | |
|---|---|
| *URL*/ | Gathers only the specified location. For example, entering **http://www.microsoft.com/** causes the Gatherer to collect all the Web pages in that directory but no pages from any of its subdirectories. In this example, http://www.microsoft.com/default.html is collected but http://www.microsoft.com/iis/roadmap.asp is not. |
| *URL*/**%*** | Gathers all pages that start with *URL*. For example, entering **http://www.microsoft.com/%*** causes the Gatherer to collect all Web pages that start with http://www.microsoft.com/ |
| *URL*/**%*:***n* | Gathers all pages *n* or fewer levels under the specified root node. In other words, entering this value specifies to gather all pages with URLs that start with *URL*/ that have *n* – 1 or fewer further slashes.<br><br>For example, entering **http://home.microsoft.com/%*:2** causes the Gatherer to collect all pages that are two or fewer levels beneath http://home.microsoft.com. In other words, this value causes Gatherer to collect Web pages whose URL match the form http://home.microsoft.com/*/* where * represents any number of characters.<br><br>Thus, in this case the Gatherer collects http://home.microsoft.com/default.html and http://home.microsoft.com/reading/news.asp but not http://home.microsoft.com/reading/more/news.asp |

**Note** A valid group specification must end with /, **%***, or **%*:***n*, where *n* is as described preceding.

In addition, you can also combine two or more of the group specifications listed preceding under a single channel name. You use a plus sign to combine group specifications; surround the plus sign with spaces so that the URLs are interpreted correctly. The format is:

*URLspec* + *URLspec* + ...

For example, if you want to deliver files from the Microsoft® MSN™ online service together with files from the Microsoft corporate Web site in a channel called MS-Total, you can specify the following:

**http://www.msn.com/%* + http://www.microsoft.com/%*:2**

**Note** When the Gatherer collects the files for a group, it collects all files that match your group specification and that can be retrieved using HTTP. Such collection includes HTML files with the extensions .asp, .htp, and .shtml, as well as binary data and image files. The Gatherer, however, does not support the HTTPS security protocol and is thus unable to gather secure Web sites.

# Using Broadcast Architecture

[This is preliminary documentation and subject to change.]

The following sections describe how to perform development tasks common when working with Broadcast Architecture:

- Displaying Video
- Writing a Custom Guide Database Loader
- Scheduling Show Reminders
- Creating TV Viewer Controls
- Writing Server Applications

These sections are closely integrated with the Broadcast Architecture sample applications provided with the Platform Software Development Kit (SDK) and the beta program for the Microsoft® Windows® 98 operating system. To locate these sample applications, see Broadcast Architecture Sample Applications.

# Displaying Video

[This is preliminary documentation and subject to change.]

To present audio and video on a computer display or to manage all the devices associated with video, you can use the Microsoft® ActiveX™ control for video in Broadcast Architecture (the Video control, Vid.ocx). You can use the Video control in applications written using Microsoft® Foundation Classes (MFC) or with the Microsoft® Visual Basic® programming system. You can also embed the Video control in the page of a World Wide Web browser or use it as a constituent control in a control that you create.

If you want, you can create your own television viewing application. Your television viewer might consist of the Video control along with an interface enabling users to select frequently viewed television channels.

The following topics describe in order implementation tasks that are necessary to use the Video control in an application, in another control, or while it is embedded in a container:

1. Getting Available Devices
2. Setting Video Input
3. Setting Video Output
4. Tuning to a Channel
5. Setting Video Priority

The Broadcast Architecture material includes four sample applications developed in different ways to demonstrate how to use the Video control. To locate these samples, see Broadcast Architecture Sample Applications.

For more information about the Video control, how to use it, and the interfaces it supports, see About the Video Control, Using the Video Control, and Video Control Reference.

# Getting Available Devices

[This is preliminary documentation and subject to change.]

Before your application can enable users to run the Microsoft® ActiveX™ control for video (the Video control) in all its possible ways, users must be aware of all of the devices available for video on their computers. A possible point at which your application can get and list available devices is when it starts. Doing so initially provides users a listing of the capabilities of their computers.

For your application to get these devices, it must first declare and allocate storage for variables of **BPCDevices** and **BPCDeviceBase** object types. Your application then assigns the collection of all the available input and output devices to the **BPCDevices** variable with the **BPCVid** object's **Devices** property and enters a loop to get each **BPCDeviceBase** object that represents an available device for video. Your application can use the **BPCVid** object's **DeviceCount** property to determine the number of times to loop. Your application obtains a string representing the name of each available video device with the **Name** property of the **BPCDeviceBase** object and adds those strings to a list box on the user interface or to an array of string elements. In applications written using MFC, you must generate the **CBPCVid**, **CBPCDevices**, and **CBPCDeviceBase** classes when you add the Video control to your application. You must also include the header files that define those classes in any source files that use them.

The best way to add initialization functionality to your application depends on how you develop your application. With the Microsoft® Visual Basic® programming system, you implement initialization in the **Form_Load** procedure. With MFC, you implement initialization in the **OnInitDialog** method of your dialog class. With a Hypertext Markup Language (HTML) document, you implement initialization in the script that runs when a Web browser loads the page.

In this release of Broadcast Architecture, the Video control only supports sending analog video to the digital VGA display surface of the control itself. Therefore, the Video control cannot set an output device; it can only set which input device to receive video from. If a user's computer has an analog television tuner card installed and your application provides a list of available devices, the friendly name of the WDM stream class minidriver for the tuner device will be listed as an input source. WDM here stands for (Microsoft®) Windows® Driver Model.To locate more information on working with display surfaces, see Further Information on Development Tasks in Broadcast Architecture.

## Setting Video Input

[This is preliminary documentation and subject to change.]

Before your application can display video, it must first assign an input device as a source for the video stream. If your application initially provides a list of names of available devices on its user interface or has these names stored in an array of strings, a user can select a device from the list or array. Then, through a command button on the user interface, the user can set the device as the input source for video.

For your application to set an input, it must first declare and allocate storage for variables of **BPCDevices** and **BPCDeviceBase** object types. Your application then assigns the collection of all the available input and output devices to the **BPCDevices** variable with the **BPCVid** object's **Devices** property and enters a loop to get each **BPCDeviceBase** object representing an available video device. Your application obtains a string representing the name of each device with the **Name** property of the **BPCDeviceBase** object and compares the string with the name of the device selected from the list or array. If the string values are the same, your application can assign the selected device as the input source with the **Input** property of the **BPCVid** object. Before setting the device as the input source, your application can confirm that the selected device is for video input with the **IsInput** property of the **BPCDeviceBase** object.

# Setting Video Output

[This is preliminary documentation and subject to change.]

In future releases of Broadcast Architecture, when the Microsoft® ActiveX™ control for video (the Video control) supports different outputs for video, your application will be able to assign an output device to receive the video stream. An application sets the video output the same way as it sets the video input, as described in Setting Video Input, except for the following differences:

- Your application assigns the selected device to be the output source with the **Output** property of the **BPCVid** object.
- To confirm that a selected device is for video output, your application should use the **IsOutput** property of the **BPCDeviceBase** object.

# Tuning to a Channel

[This is preliminary documentation and subject to change.]

To display video from a specific broadcast channel, your application must either assign a channel to the device that was set as the input source for video or tune to a channel in a specific *tuning space*. In either case, the end result is that the device tunes to the channel and the received video displays on the surface of the Microsoft® ActiveX™ control for video (the Video control). The following topics describe these two ways of tuning to a channel:

- Setting a Channel Number
- Tuning to a Channel in a Tuning Space

You can set the **BPCVid** object's **VideoOn** property in your application to either TRUE or FALSE to display video or to stop video from displaying. Setting this property is only effective when the input source is a *tuner*. Your application can stop an input source from sending video by calling the **BPCVid** object's **Close** method.

To locate more information on working with display surfaces, see Further Information on Development Tasks in Broadcast Architecture.

## Setting a Channel Number

[This is preliminary documentation and subject to change.]

To set a channel number through elements on the user interface, your application must first determine if the input source for video is from a tuner with the **HasChannel** property of the **BPCDeviceBase** object. If your application determines that the input device is a tuner with this property, your application assigns the specified channel to the tuner with the **BPCDeviceBase** object's **Channel** property. You can also use the **ChannelAvailable** method of the **BPCDeviceBase** object in your application to determine whether a tuner can tune to the specified channel before actually tuning to the channel.

## Tuning to a Channel in a Tuning Space

[This is preliminary documentation and subject to change.]

To tune to a channel in a specific tuning space, such as the tuning space for analog cable TV, through elements on the application's user interface, your application should first determine if the user's computer contains a tuner that supports that tuning space. To get information about or tune to a channel in a specific tuning space, you must first obtain the number that identifies such a tuning space. You obtain such a tuning space number from the following registry key:

**HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\TV Services\Tuning Spaces**

To obtain the description of a tuning space in the registry to determine if it is the one you want, click the tuning space number to select it. Details about the tuning space then appear in the right section of the registry's window.

To make this determination, your application should call the **BPCVid** object's **TSDeviceCount** method and specify the tuning space number. If this method returns a number greater than zero, your application can then call the **BPCVid** object's **Tune** method and specify the tuning space number, the channel number to tune to, and the default values for the video and audio subchannels. As shown in the video sample applications, the default value for both is –1, which allows the tuner to choose an appropriate subchannel.

# Setting Video Priority

[This is preliminary documentation and subject to change.]

You might want to set the priority for the Microsoft® ActiveX™ control for video (the Video control) in your application to a high level to allow it to retain control of its input device in the event that another application containing an instance of the Video control starts. On the other hand, you might not want the Video control in your application to retain control. In that case, you set its priority to a low level.

You can set the priority level of the Video control in your application with the **Priority** property of the **BPCVid** object. For more information about priority levels of the Video control, see Device Contention and **BPCDevices.Priority**.

# Writing a Custom Guide Database Loader

[This is preliminary documentation and subject to change.]

The *Guide database* stores information about what television programming is available through Broadcast Architecture applications. The information for the Guide database comes from many sources, such as the Internet, *vertical blanking interval* (VBI) transmissions, and satellite data streams. The database is a Microsoft® Jet database that can be accessed using Microsoft® Data Access Objects (DAO). To locate more information on DAO, see Further General Information.

To streamline access to the database, Broadcast Architecture uses loader programs. The two sample applications described in this section, Load and Download, show how to write custom database loader programs. To locate the source code for these sample applications, see Broadcast Architecture Sample Applications.

These sample loaders are written using Microsoft® Foundation Classes (MFC) and several libraries from Broadcast Architecture. Each sample has the same general structure and shows the important functions of a loader application. The sample called download reads programming information from a text file and adds it to the Program Guide database. This sample shows how to write a full featured loader. The sample called load, a simplified example, gives an unobstructed view of how loader functions for adding channels and shows work.

The following topics describe, in order, the tasks involved in loading data into the *Program Guide*:

1. Getting Data to the Client discusses possible sources of Program Guide data and how to get it to the *broadcast client*.
2. Making a Loader DLL describes the required features of a Guide database loader.
3. Using Guide Data Objects to Access the Database shows how to use functions in Broadcast Architecture to simplify the loader.
4. Entering Data shows specific examples of how to edit records in the Guide database.
5. Removing Records Using Special Functions describes how to use special queries in the Guide database to perform housekeeping functions.
6. Installing and Running Your Loader shows how to use the Loadstub.exe component and the *Task Scheduler* in the Microsoft® Windows® 98 operating system to run the loader.

## Getting Data to the Client

[This is preliminary documentation and subject to change.]

The first step to updating the Guide database is to get information about what television programming is on what channel. The procedure for this depends on the source of the data. For example, if the data is in a file on the Internet, you can send it over the VBI of a predetermined television channel by using

*Internet channel broadcasting*. The user can then subscribe to the appropriate Internet channel, and the file appears in the designated file directory. Then a loader can read the file and transfer the results into the Guide database.

Many other possible sources of data exist, such as e-mail and World Wide Web pages. You can even write a program that associates VCR Plus codes, used with some VCRs to specify channels and times to be recorded, with program names to make your own custom show. In this example, the data is hard-coded into the loader. Hard-coding data into the loader means you can see how the loader works without getting bogged down with the details of a particular source. To locate the source code for Load, see Broadcast Architecture Sample Applications.

# Making a Loader DLL

[This is preliminary documentation and subject to change.]

A loader is a dynamic-link library (DLL) that can be called by Loadstub.exe, a component that starts a loader application. The prototype for an entry-point function for a loader is:

```
ExitCodeList APIENTRY
EPG_DBLoad(int &argc, _TCHAR **argv, CdbDBEngine &db
    , PFNFORCEQUIT pfnForceQuit);
```

The first two arguments, *argc* and *argv*, contain the command-line arguments that Loadstub does not handle itself. You can pass any arguments you like here. For example, you can pass the user's zip code to select the channels to load for a given region.

In addition to such loader-specific arguments, Loadstub can pass a **/c** argument to request that the loader compact the database, or a **/p** argument to request a partial database update. For more information on how to handle these arguments, see Removing Records Using Special Functions.

Another command-line argument you can specify is **/L**, which tells the loader about the source of the data. The format of this argument depends on the data source. For example, if the data comes from the Internet, the argument can be the Uniform Resource Locator (URL) of the data source.

The third argument for the entry-point function, *db*, is a reference to the DAO database engine. This value must be passed to any function that accesses the Guide database. To locate more information on the **CdbDBEngine** object used in this argument, see Further Information on Development Tasks in Broadcast Architecture. The sample loader saves the value in a member of the application object called **m_pDAODB**.

The last argument, *pfnForceQuit*, is a pointer to a function that returns a value that is true if the operating system is trying to shut down the loader. Your loader should check this function from time to time to see if the loader should shut down. Doing so prevents the operating system from forcing the loader to exit without performing cleanup operations such as closing databases and releasing system resources.

## Using Guide Data Objects to Access the Database

[This is preliminary documentation and subject to change.]

It is possible to access the Guide database directly through the *CdbDBEngine* variable passed to the loader. You can avoid having to deal with the details of the database by using the Broadcast Architecture component library Dbsets.dll. This library provides objects that give access to each of the tables in the database information on shows displayed in the Program Guide. For more information, see Guide Data Objects.

The application object in the sample loader includes pointers to each of the objects defined in Dbsets.dll. The member function **OpenTables** initializes these pointers. After calling **OpenTables**, the loader can access any table in the database by using the appropriate object.

## Entering Data

[This is preliminary documentation and subject to change.]

The sample loader Load enters data into the database by using the method **Handle** in its application object. As mentioned in Getting Data to the Client, Load uses hard-coded values. Your loader should include code to read the data source in its implementation of a similar method.

The sample loader creates objects for each record to add to the database, then uses the corresponding Guide data object to update the record. Notice that several of the created objects require objects or identifiers from other objects. For example, when adding a channel you must supply the identifier of the station record. Supplying the identifier means that you must have a station record before you can create a channel record.

Be careful when using dates and times. The database stores times in coordinated universal time (UTC). You must convert this to or from local time when appropriate. You can use the member variable *m_odtsTimeZoneAdjust* to make the conversion.

While the **Handle** method is running, the loader is not receiving system messages. This means that if the system tries to shut down the application, you do not know about it. If the system terminates the loader application process, loading may stop while files are open or other system resources are in use. This may have disastrous results. To prevent the problem, you should call the **pfnForceQuit** function to see if a shutdown message is in the message queue. If this function returns TRUE, do whatever processing is required to safely terminate the function. In the sample loader, the address in

**pfnForceQuit** is stored in the application object's member variable *m_pfnForceQuit*.

## Removing Records Using Special Functions

[This is preliminary documentation and subject to change.]

There are several special functions that a loader can use on the database. These functions are stored in the database as queries. For more information about what queries are available, see Guide Database Query Reference.

The loader executes queries by calling the application object's member function **ExecuteActionQuery**. The **ClearOldEntries** member function uses these queries to remove excess records from the database. Removing excess records is important for keeping the database from growing too large.

## Installing and Running Your Loader

[This is preliminary documentation and subject to change.]

In order to run your loader, you must tell the TV Viewer component Loadstub.exe about it. You do this by adding an entry to the registry that describes your loader and passing this entry to Loadstub.exe.

Loadstub.exe searches for the entry in the following registry key:

**HKEY_LOCAL_MACHINE\Software\Microsoft\TV Services\Guide\Loaders**.

The name of the entry should be the globally unique identifier (GUID) for the loader, enclosed within curly braces ({}). If you have installed TV Viewer, you can use the entry for the default loader as an example. You must add a name string value to the new entry that gives the complete path to your loader. You can also include any parameters your loader requires in this entry. Loadstub.exe adds values for the last attempted run time, the last completed run time, and the last completed result.

Once the registry contains the new entry, you can run your loader with the following command line:

**loadstub /L:{00000000-0000-0000-0000-000000000000}**

When you have verified that your loader is working properly, you can set it to run at regular intervals. You do so by adding the preceding command line to the scheduled tasks list in the *Task Scheduler* within the Microsoft® Windows® 98 operating system.

# Scheduling Show Reminders

[This is preliminary documentation and subject to change.]

A *show reminder* is a special-purpose task scheduled in *Task Scheduler*, a feature of the Microsoft® Windows® 98 operating system. When a show reminder runs, it calls an application that displays a message informing the user that a television show is about to begin. TV Viewer is typically used to display show reminders, but you can also write custom display applications to display your show reminders. For example, your custom application can play music or animate the show reminder message.

Users can set show reminders in the TV Viewer user interface. In addition, Broadcast Architecture exposes several interfaces that you can use to programmatically set, modify, or delete show reminders. For example, you can create a control for an *enhancement* page that automatically schedules a reminder for next week's episode of a particular television program.

The following topics describe in order the basic tasks involved in creating a show reminder:

1. Getting and Working with Episode Data
2. Setting a Show Reminder
3. Setting a Record Reminder
4. Setting a Reminder that Appears in TV Viewer
5. Displaying a Show Reminder
6. Deleting a Show Reminder

The software supporting Broadcast Architecture includes the sample application Schsamp.dll, a Microsoft® ActiveX™ component that schedules a show reminder. To locate Schsamp.dll, see Broadcast Architecture Sample Applications.

To locate more information about Task Scheduler, see Further General Information.

# Getting and Working with Episode Data

[This is preliminary documentation and subject to change.]

Before you can set a show reminder, your application must obtain information about the episode or episodes for which you are scheduling the reminder. Obtaining information can be as simple as using known episode data hard-coded into a script or application, or as complicated as implementing an episode object in your application and then querying the *Guide database* to populate the object's properties.

How you handle episode data depends largely on your application and on which Broadcast

Architecture interfaces it calls to set show reminders. For example, if you are creating a show reminder from a temporary *enhancement* page, hard-coding the data into the Microsoft® Visual Basic® Scripting Edition (VBScript) script that sets the reminder may be sufficient. This is because in this case, the application only needs to set a reminder for a specific show, the enhanced show, not for shows in general.

However, if you are calling **ITVViewer::SetReminder** method to set a show reminder, you must pass in a pointer to an **IEPGEpisode** interface. (**SetReminder** is implemented by **ITVViewer**, the TV Viewer dispatch interface.) Thus, your application must implement **IEPGEpisode** and populate that interface's property values with the data for the episode in question. This population can be done using data from the Guide database.

For more information, see the following topics:

- Episode Data Format, which describes the format of the episode data you must specify when setting a reminder.
- Retrieving Episode Data from the Guide Database, which explains ways to retrieve data from the Guide database.
- The entry on the user trigger object **EnhUser**.

## Episode Data Format

[This is preliminary documentation and subject to change.]

The amount and type of episode data that your application must provide varies depending on which of the reminder-setting methods described in Setting a Show Reminder you use, either calling the **SetReminder** method of **ITVViewer**, or by adding a reminder to the **ScheduledItems** collection of Television System Services (TSS).

If you are using the **IScheduledItems::Add** method to schedule a show reminder you must provide a fully specified *show reference*. For more information, see Show Reference Format.

If you are using **ITVViewer::SetReminder** to schedule a show reminder, your application must implement the **IEPGEpisode** interface and populate its property values with data about the episode for which you are setting a reminder. For more information, see **IEPGEpisode**.

## Retrieving Episode Data from the Guide Database

[This is preliminary documentation and subject to change.]

The *Guide database* stores *Program Guide* information about future shows. You can query this database for information about an episode.

For example, when gathering data about an episode for a show reminder, your application might query the Guide database for episode records with a specific title and start date. Once you find a matching record in the database, you can fill in the rest of the episode information for the reminder, such as channel, network, and ending time, from the record's fields.

To enable applications to retrieve or set episode and other records and fields in the Guide database, Broadcast Architecture provides the Guide data objects. These objects wrap the Data Access Objects (DAO) code necessary to access the Guide database's records and fields. Specifically, you can use the **CEpisodeTRecordset** object to select and create episode records in the Guide database. The **CEpisodeTRecordset** contains a collection of **CEpisodeT** objects, each of which wraps the field information about an individual episode record.

For more information, see the following topics:

- Guide Data Objects.
- Using the Guide Data Objects.
- Episode Table in the Guide Database Schema. The Episode Table contains episode information.

To locate information on DAO, see Further General Information.

## Setting a Show Reminder

[This is preliminary documentation and subject to change.]

A *show reminder* is a task set in *Task Scheduler* that at a specific time starts an application to remind users a show is about to begin. Broadcast Architecture provides two technologies that you can use to set a show reminder, TV Viewer and Television System Services (TSS).

The process of setting a reminder that appears in the TV Viewer user interface is identical to the process that occurs when a user sets a reminder by clicking a **Remind** button in the *Program Guide* user interface displayed by TV Viewer. To set such a show reminder, your application calls the **ITVViewer** dispatch interface exposed by TV Viewer . A show reminder set using this interface is visible from the TV Viewer search page. Such a reminder also causes an icon indicating that the reminder has been set to appear in the Program Guide.

The disadvantage to using TV Viewer to set a reminder is that the programming is more complex. Your application must ensure that TV Viewer is currently running, get a reference to **ITVViewer,** and implement the **IEPGEpisode** interface. For more information on this process, see Using **ITVViewer** to Schedule a Show Reminder.

If your application is relatively simple, for example a Microsoft® ActiveX™ control or Java applet on an *enhancement* page, you may find it more convenient to use TSS to schedule your show reminders. If you want TSS-set reminders to appear in the TV Viewer user interface as icons or on the search page, you must set reminders that meet the standards described in Setting a Reminder that Appears in

TV Viewer. TV Viewer only displays reminders set by TV Viewer in these fashions.

You can, however, use TV Viewer to display a message for a TSS-set reminder to the user, regardless of whether the TSS-set show reminders meet these standards. In other words, if you specify Tvx.exe in the command line for the reminder, TV Viewer will always attempt to display a reminder message when the reminder runs.

For more information about setting a show reminder using TSS, see Using **IScheduledItems** to Schedule a Show Reminder and Setting a Reminder that Appears in TV Viewer.

## Setting a Record Reminder

[This is preliminary documentation and subject to change.]

A *record reminder* is a task set in *Task Scheduler* that at a specific time starts an application that controls a VCR or similiar device to record a show. Record reminders are set in the same manner as show reminders save for a few additional steps. These are listed below:

1. You should not specify Tvx.as the display application for a record reminder. Instead, the record reminder should use the application specified in this registry value:

   **HKLM\Software\Microsoft\TV Services\Explorer\HelperApp**

   If that value is not set, the reminder should use Tvwakeup.exe.

2. Set the TASK_FLAG_SYSTEM_REQUIRED flag for the reminder. This can be done using the Task Scheduler. This flag causes TV Viewer to tune to the channel even if the system is sleeping. Note that this does not wake up the display if the system is sleeping, so that recording can be done quietly.

**Note** The TASK_FLAG_SYSTEM_REQUIRED flag should not be set for standard show reminders. Version 1.0 of Broadcast Architecture does not support standard show reminders that go off while the system is sleeping.

3. If the record reminder has associated helper applications, for example applications that automatically tunes or turn off a VCR, these applications should be listed in the **StartRecordingApp** and/or **EndRecordingApp** values in the registry. These values appear under this registry key:

   **HKLM\Software\Microsoft\TV Services\Explorer\**

   Two minutes before the show begins, TV Viewer checks for the **StartRecordingApp** value, and if found, starts that application with the following command line syntax:

   <application> **-f** <function> **-d** <duration> **-c** <channel> **-s** <tuning_space> **-t** <title> **-r**

<show_reference>

Where

*<application>*

Is the path and filename of the application specified in either **StartRecordingApp** or **EndRecordingApp**. If recording is starting this value is the application in **StartRecordingApp**, if recording is ending, this value is the application in **EndRecordingApp**.

*<function>*

Indicates whether recording is starting or ending. If recording is starting this value is "start", if recording is ending, this value is "end".

*<duration>*

Is the duration, in minutes, of the show being recorded. The call to the end recording application always specifies a duration of 0.

*<channel>*

Is the channel number.

*<tuning_space>*

Specifies the name of the tuning space. For example, "Cable"

*<title>*

Specifies the title of the show being recorded.

*<show_reference>*

The show reference as specified in the record reminder.

Two minutes after the show ends, TV Viewer checks for the **EndRecordingApp** value, and if found, starts that application with a command line as described preceding.

For example, if you have set the value of **StartRecordingApp** to "C:\Startrec.exe" and **EndRecordingApp** to "C:\Endrec.exe" then when recording starts TV Viewer starts Startrec.exe with a command line such as the following:

```
C:\Startrec.exe -f "start" -d 30 -c 6 -s "Cable" -t "Tale Spin"
-r "1997/11/6!0/0/0!23:00!0!0!0!0!0!0!0!0!0!"!'DISN'!'Cable'!2!Tale Spin"
```

When recording ends, TV Viewer starts Endrec.exe with a command line such as the following:

```
C:\Endrec.exe -f "end" -d 0 -c 6 -s "Cable" -t "Tale Spin"
-r "1997/11/6!0/0/0!23:00!0!0!0!0!0!0!0!0!0!"!'DISN'!'Cable'!2!Tale Spin"
```

## Setting a Reminder That Appears in TV Viewer

[This is preliminary documentation and subject to change.]

TV Viewer provides a user interface that displays a information about currently set reminders. These reminders appear when the user opens the search page in TV Viewer and clicks **My Reminders** in the list of categories. In addition, a reminder icon appears in the **Program Summary** panel when the user clicks an episode in the Program Guide for which a reminder is set.

Users can use the **MyReminders** list to view, edit, or delete show reminders. The reccommended way to set reminders that appear in the TV Viewer use interface is to use **ITVViewer::SetReminder**.

However, you can use the **IScheduledItems::Add** method of TSS to set reminders that appear in TV Viewer if the reminders meet the following standards:

- Ensure that the show reference contains the start date, channel number, tuning space, station call letters and title.
- Set the the application flag to **tvviewer**. For more information, see Using IScheduledItems to Schedule a Show Reminder.
- Only set one-time, weekly or daily reminders.
- If it is a record reminder, follow the procedures described in Setting a Record Reminder.
- Limit the number of reminders you add. The reminder list displayed by TV Viewer has a limit of fifty reminders.
- Do not add duplicate reminders that specify the same show reference.
- Set the reminder to run no more than 30 minutes before the show starts.

## Displaying a Show Reminder

[This is preliminary documentation and subject to change.]

A *show reminder* is simply a task set in *Task Scheduler*. As such, it does not provide a user interface. When it runs, it calls a display application, such as TV Viewer, to parse its *show reference* and display a reminder message to the user. When that display application is TV Viewer, the application also offers the user the opportunity to tune immediately to the channel on which the episode is broadcast. Your application specifies which display application to use when it sets the reminder.

If your application requires a custom reminder dialog box or special functionality, you can write a custom display application to handle show reminders. For example, you can write an application that displays an image of the show's cast and plays the theme song when it reminds the user the show is about to begin.

To be compatible with Broadcast Architecture show reminders, a custom display application should accept a show reference as input. It should then parse the show reference to obtain the episode data and display the reminder to the user. For more information on show references, see Show Reference Format.

To locate more information on Task Scheduler, see Further General Information.

## Deleting a Show Reminder

[This is preliminary documentation and subject to change.]

Show reminders are deleted during the course of successful completion. When a show reminder runs, unless it is a recurring show reminder, the Task Scheduler deletes the reminder from the Tasks folder.

In addition, show reminders can also be deleted by the following:

- You can delete a show reminder programmatically using either **ITVViewer::DeleteReminder** or by removing the item corresponding to the reminder from the **IScheduledItems** collection.
- Invalid show reminders, those having an invalid show reference or incorrect command line syntax, are deleted automatically by TV Viewer.
- If the show reminder is one that appears in the TV Viewer reminders list, you can use the TV Viewer user interface to delete a show reminder.
- You can delete a show reminder by opening the Tasks folder, typically C:\Windows\Tasks, right-clicking the task for the reminder, and selecting Delete.

**Note** Reminders are scheduled as hidden tasks. Therefore they will not appear in the Tasks folder unless you have set your viewing options to show hidden files. To do this, go to the Tasks directory, typically C:\Windows\Tasks, and type `attrib –h *.*`.

# Creating TV Viewer Controls

[This is preliminary documentation and subject to change.]

TV Viewer, a component of the Microsoft® Windows® 98 operating system, is the user interface for Broadcast Architecture. TV Viewer hosts several controls, such as Internet Explorer and the Microsoft® ActiveX™ control for video (the Video control), that it uses to present data such as live video, *Program Guide* information, and *enhancements*.

You can extend the Broadcast Architecture user interface by writing custom controls that are hosted by TV Viewer. Because Broadcast Architecture uses a layout based on *Hypertext Markup Language* (HTML) for its user interface, writing a control for TV Viewer is as simple as writing an ActiveX control.

However, you need not stop there. TV Viewer exposes COM interfaces that enable your control to interact with it. You can query TV Viewer about its state, programmatically control it, and receive event notifications from it. You call the TV Viewer interfaces from any programming language that supports COM and the **GetActiveObject** Automation function.

For example, you can write a control that queries the state of TV Viewer to monitor a user's viewing habits. This information might be used locally to build a user profile and then to recommend future broadcasts that the user might enjoy. The information might also be transmitted through the *back channel* to advertisers and broadcast content providers, to enable them to compile information about which episodes are most successful.

The following topics describe tasks your TV Viewer control can implement:

- Connecting to TV Viewer
- Controlling TV Viewer
- Receiving Events from TV Viewer

The software supporting Broadcast Architecture includes two sample applications that work with TV Viewer. To locate these samples, see Broadcast Architecture Sample Applications.

For more information about TV Viewer and the interfaces its exposes, see TV Viewer and the **ITVViewer** and **ITVControl** interfaces.

# Some Useful Objects

[This is preliminary documentation and subject to change.]

Brtest is a library that includes several objects that applications can use to send data to the MMR. The following table lists the objects:

| **Object** | **Description** |
|---|---|
| **CData** | Base class for data storage |
| **CString** | **CData**-derived class for strings |
| **CSession** | Base class for connections to the MMR |
| **CMulticast** | Multicast session |
| **CTunnel** | Tunneling session |

## CData

[This is preliminary documentation and subject to change.]

The **CData** class is a base class for objects that contain data. It includes member functions for locking the data in order to control access to the data in multithreaded applications.

## CString

[This is preliminary documentation and subject to change.]

The **CString** class is derived from the **CData** class. The data in this object is a null-terminated string.

## CSession

[This is preliminary documentation and subject to change.]

The **CSession** class is an abstract base class for a connection to the MMR. This class should not be used directly. Instead, use **CMulticast** or **CTunnel** depending on the type of connection.

## CMulticast

[This is preliminary documentation and subject to change.]

The **CMulticast** class is derived from **CSession**. It includes a method to broadcast multicast packets that the MMR can route to an output driver.

## CTunnel

[This is preliminary documentation and subject to change.]

The **CTunnel** class is derived from **CSession**. The constructor opens a socket connection to the MMR called a tunnel. The send method sends data through this tunnel to the MMR. Using a tunnel allows the MMR to control the rate that the server application sends the data.

# Key Routines in Wsend

[This is preliminary documentation and subject to change.]

The Wsend program uses the objects in the Brtest library to send data to the MMR. This program can use either tunneled access or send IP multicast packets.

Much of the code in Wsend.cpp is meant to handle saving and restoring data from the registry and also to handle the user interface. The operation of these functions is not covered in this chapter.

The functions that illustrate how to connect to the MMR and send data are described in Multicasting Functions and Tunneling Functions.

## Multicasting Functions

[This is preliminary documentation and subject to change.]

When you start a multicast session in WSend, it calls the **OnStartMulticast** function. This function starts a thread that multicasts data using the **MulticastWorkerThread** function.

The **MulticastWorkerThread** function runs until you stop the session. The **OnStopMulticast** function handles stopping the session by setting an abort event, waiting for the event to be handled by the thread, then shutting down the thread.

The **MulticastWorkerThread** function handles sending the multicast packets. It calls the **BandwidthThrottleSend** function repeatedly to send the data. The **BandwidthThrottleSend** function computes the speed for sending data based on the amount of data to send and the bandwidth of the output driver. The result of the speed computation is the number of milliseconds to wait between send operations. While waiting, **BandwidthThrottleSend** checks for abort events, to know when to stop sending.

## Tunneling Functions

[This is preliminary documentation and subject to change.]

When you open a tunnel session in WSend it calls the **OnOpenTunnel** function. This function opens a tunnel to the MMR and, if the tunnel opens without error, starts a thread that sends data through the tunnel. The **OnCloseTunnel** function stops the transfer by setting an abort event, waiting for the thread to process the abort event, and shutting down the thread.

The **TunnelWorkerThread** function is nearly identical to the **MulticastWorkerThread** function. The difference is that the send function in BandwidthThrottleSend uses a tunnel to send the data to the MMR instead of broadcasting multicast packets.

# Key Routines in Wlisten

[This is preliminary documentation and subject to change.]

Wlisten.exe waits for specified multicast packets and displays information about them. The key function in this program is **WorkerThread**, a routine that runs as a thread when WListen is listening for multicast packets.

**WorkerThread** starts by creating an event that indicates when the socket is available for reading. The main loop of the thread waits for this event or a completion event. If it sees the read event, **WorkerThread** reads data from the socket and processes the data so it can be displayed. When the user shuts down Wlisten, the program sets a completion event that causes **WorkerThread** to close down and exit.

# Numbers

[This is preliminary documentation and subject to change.]

**802.3 intermediate driver**

A Network Driver Interface Specification (NDIS) intermediate driver that is a component of Broadcast Architecture. This driver translates Multipacket Transport (MPT) packets from satellite networks into Internet Protocol (IP) packets. This translation makes it possible for Windows Sockets (WinSock) to handle the data as it does any other data carried by Transmission Control Protocol/Internet Protocol (TCP/IP).

# Numbers

[This is preliminary documentation and subject to change.]

**802.3 intermediate driver**
: A Network Driver Interface Specification (NDIS) intermediate driver that is a component of Broadcast Architecture. This driver translates Multipacket Transport (MPT) packets from satellite networks into Internet Protocol (IP) packets. This translation makes it possible for Windows Sockets (WinSock) to handle the data as it does any other data carried by Transmission Control Protocol/Internet Protocol (TCP/IP).

# A

[This is preliminary documentation and subject to change.]

**ActiveX control**
: A reusable software component that can quickly add specialized functionality to World Wide Web sites, desktop applications, development tools, and enhanced video programs. These controls typically provide user interface elements, such as buttons. Microsoft® ActiveX™ controls were previously known as OLE custom controls or OCXes.

**Address Resolution Protocol**
: (ARP) The protocol that maps Internet Protocol (IP) addresses to the physical hardware addresses of specific networks. This protocol operates at the level of network operating systems and is not generally accessible to applications.

**ADSL**
: *See* Asymmetrical Digital Subscriber Line.

**analog television tuner card**
: A type of broadcast receiver card that receives and processes analog video signals.

**analog video**
: Video transmitted by an analog signal, such as an NTSC, PAL, or SECAM transmission.

**announcement filter**
: In the Announcement Listener, one of a collection of dynamic-link libraries (DLLs) or executable files that act as Automation servers. An announcement filter distinguishes between announcements that are of interest to a particular broadcast client's user and those that are not. Multiple filters can exist for different kinds of data; for example, in Broadcast Architecture Internet channel broadcasting and enhancements have their own filters.

**Announcement Listener**
: A operating-system service that receives announcements of upcoming data. Using Announcement Listener, the broadcast client can filter incoming data at the network interface (that is, at the broadcast receiver card) so the client only receives data of interest.

**API**
: *See* application programming interface.

**application**

A computer program designed to help people perform a certain type of work. An application differs from an operating system (which runs a computer), a utility (which performs maintenance or general-purpose chores), and a programming language (with which computer programs are created). An application can manipulate text, numbers, graphics, or a combination of these elements. In the context of Broadcast Architecture, potential applications might enable users to shop at home, find information, order movies, and so on.

**application layer**

The seventh and highest layer in the International Organization for Standardization's Open Systems Interconnection (OSI) model. The application layer contains the signals sent during interaction between user and application and that perform useful work for the user, such as file transfer. *See also* data link layer, transport layer.

**application programming interface**

(API) An interface exposed by a software module as a means for other software modules to interact with it. For example, applications generally interact with an operating system by way of the APIs the operating system exposes. In the case of Microsoft® Windows® 98 and Microsoft® Windows NT®, the APIs consist of functions that an application uses to request operating-system services, such as screen management, keyboard input, printer output, and so forth.

**ARP**

*See* Address Resolution Protocol.

**ASCII**

(American Standard Code for Information Interchange) A coding scheme that assigns numeric values to letters, numbers, punctuation marks, and certain other characters. By standardizing the values used for these characters, ASCII enables computers and computer programs to exchange information. Although it lacks accent marks, special characters, and non-Roman characters, ASCII is the most universal character-coding system.

**Asymmetrical Digital Subscriber Line**

(ADSL) A technology that allows data to be sent over ordinary twisted-pair telephone lines at T1 trunk-line speeds, currently up to 1.544 megabits per second, and to be returned at a current rate of 16 kilobits per second. ADSL provides a bidirectional voice signal while data is sent.

**authorization key**

A Triple DES key used to encrypt session keys.

## B

[This is preliminary documentation and subject to change.]

**back channel**

The segment of a two-way communications system that flows from the consumer back to the content provider, or to a system component, to provide feedback.

**backbone**

The top level in a hierarchical network.

**bandwidth**

Literally, the frequency range of an electromagnetic signal, measured in hertz (cycles per second). The term has come to refer more generally to the capacity of a channel to carry information, as measured in data transferred per second. Transfer of digital data, for example, is measured in bits per second.

**bandwidth reservation**

The process of setting aside bandwidth on a specific broadcast channel for a specific data transmission. A content server application reserves bandwidth on a Microsoft Multicast Router by calling the **msbdnReserveBandwidth** function. This function forwards the request to a bandwidth reservation server. The server returns a unique reservation identifier if the bandwidth can be reserved. *See also* IP multicast address assignment.

**baseband**

Describes transmissions using the entire spectrum as one channel. Alternatively, baseband describes a communication system in which only one signal is carried at any time. An example of the latter is S-video or a composite video signal that is not modulated to a particular television channel. *See also* broadband.

**baud**

Number of bits per second, a measure of data-transmission speed. Baud was originally used to measure the transmission speed of telegraph equipment but now most commonly measures modem speed. The measurement is named after the French engineer and telegrapher Jean Maurice-Emile Baudot.

**broadband**

Describes high-frequency transmissions over coaxial cable or optical fibers, involving sending several data streams simultaneously. Broadband is also sometimes used to describe high-speed networks in general. *See also* baseband.

**broadcast**

In general terms, a transmission sent simultaneously to more than one recipient. In Internet terminology, a transmission sent to a single address to be forwarded to many recipients. In practice, Internet broadcasts only function on local networks, because routers do not forward them. Broadcast Architecture uses a refinement of this Internet technique known as multicast, in which routers forward transmissions. In multicast, each transmission is assigned its own Internet Protocol (IP) multicast address, allowing clients to filter incoming data for specific packets of interest at the network interface card.

**Broadcast Architecture**

The Microsoft computer software and hardware design that enables personal computers to serve as clients of broadband digital and analog broadcast networks.

**Broadcast Architecture subsystems**

The functional parts that make up Broadcast Architecture. These include the Broadcast and Data Receiver subsystem, the Data Services subsystem, the Video and Display subsystem, the User Interface subsystem, and the Television Client Services subsystem. For more information, see the Broadcast Client Architecture section of the Broadcast Architecture Programmer's Reference.

**broadcast client**

A versatile personal computer that can receive and display broadband digital and analog broadcasts, blending television with new forms of information and entertainment. Broadcast client programming can include television, audio, World Wide Web pages, and computer data content.

**Broadcast Cryptography API**

Broadcast Architecture functions that help CryptoAPI decrypt content using encryption keys.

**broadcast data encoder**

A hardware subsystem provided by an independent hardware vendor that encodes data for broadcast, for example pay-per-view shows that only subscribers may watch. A broadcast data encoder receives data streams from the Microsoft Multicast Router through an output driver.

**broadcast receiver card**

A printed circuit board or adapter that can be plugged into a computer to receive and process broadcast signals, such as television or other broadcast data. One type of broadcast receiver card is a satellite receiver card. Another is an analog television tuner card.

**broadcast server**

A computer that sends broadcast programming across a broadcast channel to broadcast clients, in some cases forwarding data over the Microsoft Multicast Router. The programming sent can include television, audio, World Wide Web pages, and digital data such as stock prices, multimedia magazines, and computer software.

**Broadcast Architecture transport**

A Broadcast Architecture component that manages broadcast data received through a broadcast receiver card. The Broadcast Architecture transport binds to the Broadcast Architecture NIC miniports, which control and retrieve data in formats such as MPEG, audio, Microsoft Broadcast Data Network (MSBDN), or any arbitrary format.

## C

[This is preliminary documentation and subject to change.]

**CDF**

*See* Channel Definition Format.

**CERN**

(Conseil Europeen pour le Recherche Nucleaire, or European Laboratory for Particle Physics), a research laboratory with headquarters in Geneva, Switzerland. CERN pioneered work in developing the World Wide Web. CERN intended the Web to help scientists share information.

**channel**

In general, a path or link through which information passes between two devices. For example, a television channel carries a specific sequence of television programming. In Microsoft® Internet Explorer version 4.0, a channel is a subscription to a World Wide Web site defined by means of a Channel Definition Format (CDF) file.

**Channel Definition Format**

(CDF) A specification developed by Microsoft and presented to the World Wide Web Consortium (W3C) that allows applications to send World Wide Web pages to users. Once a user subscribes to a CDF channel, any software that supports the CDF format automatically receives any new content posted on the channel's Web server. The default client subscription application for Internet channel broadcasting in Broadcast Architecture stores subscription information as CDF files.

**chroma**

The color portion of the video signal that includes hue and saturation information. Hue refers to a tint or shade of color. Saturation indicates the degree to which the color is diluted by luminance or illumination. *See also* YUV.

**class**

In general terms, a category. In programming languages, a class is a means of defining the structure of one or more objects. *See also* device class.

**class driver**

A standard driver provided with the operating system that provides hardware-independent support for a given class of devices. Such a driver communicates with a corresponding hardware-dependent minidriver using a set of device control requests defined by the operating system. These requests are specific to the particular device class. A class driver can also define additional device control requests itself. A class driver provides an interface between a minidriver and the operating system.

**client**

Generally, one of a group of computers that receive shared information sent by a computer called a server over a broadcast or point-to-point network. The term client can also apply to a software process, such as an Automation client, that similarly requests information from a server process and that appears on the same computer as that server process, or even within the same application.

**client subscription application**

A client subscription application provides an interface that enables a user to subscribe or cancel a subscription to one or more channels in Internet Explorer 4.0.

**closed captioning**

Real-time, written annotation of the currently displayed audio content. Closed captioning usually provides subtitle information to hearing-impaired viewers or to speakers of a language other than that on the audio track.

**color keying**

A display technique in which a selected Video Graphics Array (VGA) color is replaced with video wherever that color appears on the screen. For example, television news programs commonly use color keying to replace a blue backdrop mounted behind a weather announcer with a video picture of a weather map.

**COM**

*See* Component Object Model.

**commit**

To allocate a device data stream and tune a specified connection to that device data stream. Such commitment is performed by a NIC miniport.

**common library interface functions**

Software routines supplied by Broadcast Architecture that add network driver functionality to a broadcast receiver card and manage computer memory resources for hardware-specific code. To use these common library interface functions, a hardware vendor building a NIC miniport should link the miniport code to the common library.

**Component Object Model**

(COM) An object-oriented programming model for building software applications made up of modular components. COM allows different software modules, written without information about each other, to work together as a single application. COM enables software components to access software services provided by other components, regardless of whether they involve local function calls, operating system calls, or network communications.

**content producer**

A person or company creating broadcast content. Content can include television programming, data, World Wide Web sites, and software applications.

**content provider**

A person or company delivering broadcast content. Content can include television programming, data, World Wide Web sites, and software applications.

**content server application**

An application written by a content provider and running on a computer at the broadcast head end that gathers, schedules, and sends data to the appropriate Microsoft Multicast Router. For more information, see the Writing Content Delivery Software section of the Broadcast Architecture Programmer's Reference.

**content subnet**

A network located at a broadcast head end that connects one or more content server applications to one or more Microsoft Multicast Routers.

**CPU**

Central processing unit. The computational and control unit of a computer; this device, usually a single chip, interprets and executes instructions. Examples include the Intel Pentium processor.

**CRC**

*See* cyclic redundancy check.

**CryptoAPI**

An application programming interface (API) that provides an abstraction layer for encryption and decryption services provided by a cryptographic service provider. Because CryptoAPI furnishes this layer, applications can use different encryption methods without requiring information about the hardware or software involved. This API also provides a way of protecting sensitive key data. *See also* Broadcast Cryptography API.

**cryptographic service provider**

(CSP) An independent software module that contains cryptography algorithms or services that are integrated into CryptoAPI. Many CSPs are Microsoft® Win32® application programming interface (API) service programs, managed by the Win32 service control manager. Some, such as a smart card or secure coprocessor, reside in hardware.

**CSP**

*See* cryptographic service provider.

**cyclic redundancy check**

(CRC) A common technique for detecting errors in data transmission. In CRC error checking, the sending device calculates a number based on the data transmitted. The receiving device repeats the same calculation after transmission. If both devices obtain the same result, it is assumed the transmission was error-free. The procedure is known as a redundancy check because each transmission includes not only data but additional, redundant values for error checking.

## D

[This is preliminary documentation and subject to change.]

**Data Encryption Standard**

(DES) A standard defined by the National Bureau of Standards for encryption of digital data transmissions within the United States.

**data link layer**

The second of the seven layers in the International Organization for Standardization's Open Systems Interconnection (OSI) model for standardizing computer-to-computer communications. The data link layer is one level above the physical layer. It is involved in packaging and addressing information and in controlling the flow of separate transmissions over communications lines. The data link layer is the lowest of the three layers (data link, network, and transport) that help move information from one device to another. *See also* transport layer.

**data service**

A mechanism provided by a service provider for sending broadcast data to broadcast clients. Such data can include Program Guide information, World Wide Web pages, software, and other digital information. The data service mechanism can be any broadcast process, including Internet channel broadcasting.

**data stream**

*See* stream.

**datagram**

One packet of information and associated delivery information, such as the destination address, that is routed through a packet-switching network. In a packet-switching network, data packets are routed independently of each other and may follow different routes and arrive in a different order from which they were sent. An Internet Protocol (IP) multicast packet is an example of a datagram.

**DBS**

*See* direct broadcast satellite.

**DES**

*See* Data Encryption Standard.

**DES decryptor**

A hardware device that converts cipher text encrypted to the Data Encryption Standard (DES) back to plain text.

**device**

A unit of hardware, for example an audio adapter. For hardware used with the Microsoft® Windows® 98 operating system, such a unit can be detected by Plug and Play. *See also* device class, device driver, and device object.

**device class**

A group into which devices and buses are placed for the purposes of installing and managing device drivers and allocating resources. The hardware registry tree is organized by device class. Windows 98 uses class installers to install the drivers for the different classes of hardware.

**device driver**

A software component that allows an operating system to communicate with one or more specific hardware devices attached to a computer.

**device object**

A programming object used to represent a physical, logical, or virtual hardware device whose device driver has been loaded into the operating system.

**direct broadcast satellite**

(DBS) A satellite communications technology that allows use of a very small (18 inches to 3 feet in diameter) receiver dish packaged as a consumer electronics product, enabling consumers to directly receive satellite television signals.

**DirectShow**

The Microsoft® DirectShow™ (formerly Microsoft® ActiveMovie™) application programming interface (API) is a multimedia technology designed to play video, audio, and other multimedia streams in a variety of formats that are stored locally or acquired from Internet servers. DirectShow relies on a modular system of pluggable components called filters arranged in a configuration called a filter graph. A component called the filter graph manager oversees the connection of these filters and controls the data flow of the stream.

**DirectShow filter**

A Microsoft® DirectShow™ component that processes data streams. Each filter handles part of the operation involved in receiving, decoding, transforming, scheduling, and displaying interdependent video, audio, or other data streams. Filters connect to each other in a configuration called a filter graph.

A DirectShow filter is a user-mode entity that is an instance of a Component Object Model (COM) object, usually implemented by a dynamic-link library (DLL). A Broadcast Architecture DirectShow filter can be a source filter, a transform filter, a renderer filter, or a utility filter.

*See also* KSProxy filter.

**DLL**
*See* dynamic-link library.

**downstream**
One-way data flow from head end to broadcast client. *See also* upstream.

**dynamic-link library**
(DLL) A file with the file name extension .dll that contains one or more functions compiled, linked, and stored separately from the computing processes that use them.

## E

[This is preliminary documentation and subject to change.]

**encryption key**
A specific value used by an encryption algorithm to encode and decode data.

**enhancement**
A multimedia element, such as a hypertext link to a World Wide Web page, a graphic, a text frame, a sound, or an animated sequence, added to a broadcast show or other video program. Many such elements are based on Hypertext Markup Language (HTML).

**episode**
A discrete narrative portion of an ongoing television or radio show, usually viewed in one continuous showing. Generally, an episode forms a coherent story in itself.

## F

[This is preliminary documentation and subject to change.]

**field**
In broadcast television, one of two sets of alternating lines in an interlaced video frame. In one field, the odd-numbered lines of video are drawn; in the other, the even-numbered lines are drawn. When interlaced, the two fields combine to form a single frame of on-screen video.

**File Transfer Protocol**

(FTP) A protocol that supports file transfers to and from remote systems on a network using Transmission Control Protocol/Internet Protocol (TCP/IP), such as the Internet. FTP supports several commands that allow bidirectional transfer of binary and ASCII files between systems. *See also* Hypertext Transport Protocol (HTTP).

**File Transfer Service**

(FTS) A component of Microsoft® NetShow™ server that can send files using a multicast transfer mechanism that includes forward error correction. NetShow is a component of Microsoft Site Server.

**filter**

*See* announcement filter, DirectShow filter, or KSProxy filter.

**filter graph**

A connected set of DirectShow filters that processes media data by controlling KSProxy filters.

**filter graph manager**

An object that controls how a filter graph is assembled and how data is moved through it. Applications can use the filter graph manager implicitly, allowing it to construct an appropriate filter graph for a specified media format. Applications can also access the filter graph manager explicitly — for example, to add a proprietary filter to a filter graph.

**forward error correction**

A system of error correction that incorporates redundancy into data so transmission errors can, in many cases, be corrected without requiring retransmission.

**frame**

In broadcast television, a single screen-sized image that can be displayed in sequence with other slightly different images to animate drawings. For NTSC video, a video frame consists of two interlaced fields of 525 lines; NTSC video runs at 30 frames per second. For PAL or SECAM video, a video frame consists of two interlaced fields of 625 lines; PAL and SECAM video runs at 25 frames per second. In comparison, film runs at 24 frames per second.

**FTP**

*See* File Transfer Protocol.

**FTS**

*See* File Transfer Service.

## G

[This is preliminary documentation and subject to change.]

**genre**

In Broadcast Architecture, a category of broadcast programs, typically related by style, theme, or format, for example movies or television series.

**global announcement**

A general message that is sent to all broadcast clients. Global announcements alert clients that information of a specified type will be broadcast at a certain Internet Protocol (IP) address at a certain time, or that a particular type of service is available. Global announcements are intended to be transmitted on all possible channels, so they can be received regardless of what channel a client is tuned to. Such announcements are sent relatively infrequently and may be human-readable. *See also* local announcement.

**guaranteed bandwidth**
Bandwidth that is reserved only if the requested bandwidth is available for the requested period. Once reserved, such bandwidth can be relied upon to be available. *See also* opportunistic bandwidth.

**Guide database**
In Broadcast Architecture, the database in which Program Guide information is maintained. This database is stored on the broadcast client.

## H

[This is preliminary documentation and subject to change.]

**hardware-specific interface functions**
A set of routines that a hardware vendor must implement to add broadcast network functionality.

**HDTV**
*See* high-definition television.

**head end**
The origin of signals in a terrestrial, cable, satellite, or network broadcast system. In Broadcast Architecture, the server infrastructure that gathers, coordinates, and broadcasts data is generally located at the broadcast head end.

**high-definition television**
(HDTV) Television that is delivered at a higher screen resolution than NTSC, PAL, SECAM, or other existing standards provide for.

**HTML**
*See* Hypertext Markup Language.

**HTTP**
*See* Hypertext Transport Protocol.

**Hypertext Markup Language**
(HTML) A markup language used to create hypertext documents that are portable from one platform to another. HTML files are text files with embedded codes, or markup tags, that indicate formatting and hypertext links. HTML is used for formatting documents on the World Wide Web.

**Hypertext Transport Protocol**
(HTTP) The underlying, application-level protocol by which World Wide Web clients and servers communicate on the Internet. *See also* File Transfer Protocol.

## I

[This is preliminary documentation and subject to change.]

**ICP**
Independent content provider. *See* content provider.

**IHV**
*See* independent hardware vendor.

**independent hardware vendor**
(IHV) A person or company that develops and sells hardware, in this case for Broadcast Architecture.

**independent software vendor**
(ISV) A person or company that develops and sells software, in this case for Broadcast Architecture.

**infrared**
(IR) Defines a spectrum of electromagnetic radiation with frequencies in the spectrum less than those of visible light. Remote control units usually communicate with home televisions and VCRs by using infrared signals.

**integrated receiver/decoder**
(IRD) A subscriber terminal, such as the set-top box used for satellite television systems. In the case of Broadcast Architecture, the IRD is the broadcast receiver card.

**Integrated Services Digital Network**
(ISDN) A type of phone line used to enhance data transmission speed. Data can be transmitted over ISDN lines at speeds of 64 or 128 kilobits per second, whereas standard phone lines generally limit modems to top speeds of 20 to 30 kilobits per second. An ISDN line must be installed by the phone company.

**interface**
In computing in general, the point where two elements connect so that they can work with one another, for example the connection between an application and an operating system or between an application and a user (the user interface).

In C++ programming, a collection of related methods exposed by a given class of objects. The methods in an interface are procedures that can be performed on or by those objects.

The Component Object Model (COM) architecture has become the foundation for interfaces that work with the Microsoft® Windows® 98 and Microsoft Windows NT operating systems. In earlier programming for Windows, an application programming interface (API) consisted of functions that one piece of software could call to gain access to services provided by another piece of software.

**interlacing**
A video display technique, used in current analog televisions, in which the electron beam refreshes (updates) all odd-numbered scan lines in one field and all even-numbered scan lines in the next. Interlacing takes advantage of both the screen phosphor's ability to maintain an image for a short time before fading and the human eye's tendency to average subtle differences in light intensity. By refreshing alternate lines, interlacing halves the number of lines to update in one screen sweep. An alternative video display technique, used in computer monitors, is progressive scanning. In progressive scanning, the image is refreshed one line at a time.

**Internet**
Generically, a collection of networks interconnected with routers. "The Internet" is the largest such collection in the world. The Internet has a three-level hierarchy composed of backbone networks, midlevel networks, and stub networks.

**Internet channel broadcasting**

A technology to gather and redistribute World Wide Web content. Applications that use Internet channel broadcasting package a series of Web pages or sites and deliver them to broadcast clients at regular intervals. They can provide this content to a broadcast server for retail delivery to subscribing clients, or they can provide the content on a local area network (LAN).

**Internet Protocol**

(IP) The primary network layer of Internet communication, responsible for addressing and routing packets over the network. IP provides a best-effort, connectionless delivery system that does not guarantee that packets arrive at their destination or that they are received in the sequence in which they were sent. *See also* Transmission Control Protocol/Internet Protocol and User Datagram Protocol/Internet Protocol.

**IP**

*See* Internet Protocol.

**IP multicast address assignment**

The process by which transient Class D Internet Protocol (IP) multicast addresses are allocated. Each address must be assigned uniquely across the system for a specific time period. Once that time period elapses, the address returns to the available pool. In Broadcast Architecture, these addresses are used to identify data streams being broadcast, both at the head end and in broadcast client systems in the home.

**IR**

*See* infrared.

**IR/D**

*See* integrated receiver/decoder.

**ISDN**

*See* Integrated Services Digital Network.

**ISV**

*See* independent software vendor.

## J

[This is preliminary documentation and subject to change.]

**Java**

A object-oriented, platform-independent computer programming language developed by Sun Microsystems. The Applet subclass of Java can be used to create Internet applications.

## K

[This is preliminary documentation and subject to change.]

**kernel mode**

Software processing that occurs at the level of the operating system closest to the computer, ring 0. Kernel-mode software resides in protected memory at all times and provides basic operating-system services. Kernel-mode software may activate hardware directly or interface to another software layer that drives hardware. *See also* user mode.

**KSProxy filter**

A WDM streaming component that processes data streams. A KSProxy filter is a kernel-mode entity usually implemented by a driver within the operating system. *See also* DirectShow filter.

## L

[This is preliminary documentation and subject to change.]

**local announcement**

A specific message that is intended to be sent over a particular broadcaster's own transponder. Most local announcements are not human-readable. Local announcements are generally sent at frequent intervals a short time before the broadcast they announce. *See also* global announcement.

**local area network**

(LAN) A network dispersed over a relatively limited area and connected by a communications link that enables each device on the network to interact with any other. *See also* wide area network.

**log on**

To provide a user name and password that identifies you to a computer network.

**luminance**

A measure of the degree of brightness or illumination radiated by a given source. Alternatively, the perceived brightness component of a given color, as opposed to its chroma. *See also* YUV.

## M

[This is preliminary documentation and subject to change.]

**MBONE**

*See* multicast backbone.

**merchant server**

A server that enables user purchases in a secure fashion.

**Microsoft Broadcast Data Network**

(MSBDN) A high-bandwidth broadcast network, capable of sending 2 to 25 megabits per second, or 22 to 270 gigabits per day. MSBDN is designed to use strong Data Encryption Standard (DES) encryption to secure valuable goods and services so that only paying subscribers have access to them. This high level of security will make MSBDN a suitable channel for delivery of software and other expensive digital goods and services that would otherwise be vulnerable to theft.

**Microsoft Multicast Router**

(MMR) A component that enables a content server to send a data stream to a multiplexer or other broadcast output device. The MMR calls an output driver to package and transmit a stream at the appropriate rate and in the appropriate packet format.

**miniport**

Software that communicates with a specific piece of peripheral hardware through a port. A miniport translates all applicable commands that come through the port from the computer into the appropriate hardware commands. This miniport functionality means the port does not require extensive information about each piece of hardware it supports. *See also* NIC miniport.

**miniport driver**

*See* miniport.

**minidriver**

A hardware-specific dynamic-link library (DLL) that uses a Microsoft-provided class driver to accomplish most actions and provides only device-specific controls. In Windows Driver Model (WDM), the minidriver registers each device with the class driver, which creates an object for each device. The minidriver uses these device objects to make calls to the operating system.

**MMDS**

*See* Multichannel Multipoint Distributed System.

**MMR**

*See* Microsoft Multicast Router.

**MPEG**

(Motion Pictures Experts Group standard) MPEG-1 is a standard designed for video playback of NTSC quality from CD-ROM. MPEG-1 provides video and audio compression at rates up to 1.8 megabits per second. MPEG-2 provides higher video resolutions and interlacing for broadcast television and high-definition television (HDTV). Both standards were created by the Motion Pictures Experts Group, an International Standards Organization/International Telegraph and Telephone Consultative Committee (ISO/CCITT) group set up to develop motion video compression standards.

**MPT**

*See* Multipacket Transport.

**MSBDN**

*See* Microsoft Broadcast Data Network.

**MTS**

A stereo encoding standard used with analog audio and video transmissions.

**multicast**

A point-to-many networking model in which a packet is sent to a specific address, and only those computers that are set to receive information from this address receive the packet. On the Internet, the possible IP multicast addresses range from 224.0.0.0 through 239.255.255.255. Computer networks typically use a unicast model, in which a different version of the same packet is sent to each address that must receive it. The multicast model greatly reduces traffic and increases efficiency on such networks.

**multicast backbone**

(MBONE) A virtual, multicast-enabled network that works on top of the Internet. The most popular application for the MBONE is video conferencing, including audio, video and whiteboard conferencing. However, the essential technology of the MBONE is simply multicast — there is no special support for continuous media such as audio and video. The MBONE has been set up and maintained on a cooperative, volunteer basis.

**Multichannel Multipoint Distributed System**

(MMDS) Also known as wireless cable, this system broadcasts terrestrial or satellite microwave transmissions directly to consumers' homes. The Federal Communications Commission (FCC) invented this system in the 1980s to provide competition to traditional cable television companies. There are currently 33 MMDS channels allocated, most in the 2.5-gigahertz band.

**multihomed computer**

A computer connected to two or more networks or having two or more addresses on one network. For example, a network server may be connected to multiple local area networks (LANs). Generally, a multihomed computer may send and receive data over any of its connections but does not necessarily route traffic for other computers.

**multimedia**

Online material that combines text and graphics with sound, animation, or video, or some combination of the three.

**Multipacket Transport**

(MPT) A data link adaptation layer developed by Microsoft that resides above the data link layer of a network with large, fixed-size packets, such as a satellite system. MPT allows higher-layer protocols such as Internet Protocol (IP) to exist independently of, and be unaffected by, the underlying network.

**multiplexer**

In general terms, a device for funneling several different streams of data over a common communications line. In the case of Broadcast Architecture, a multiplexer combines multiple television channels and data streams for a single transponder.

## N

[This is preliminary documentation and subject to change.]

**NABTS**

*See* North American Basic Teletext Specification.

**NDIS**

*See* Network Driver Interface Specification.

**NDIS extension**

An addition to the Network Driver Interface Specification (NDIS) that provides supplementary functionality. Broadcast Architecture supplies NDIS extensions that must be supported by all NIC miniports written to control broadcast receiver cards and by all transports that work with and access broadcast receiver cards.

**network**

In computing, a data communications system that interconnects a group of computers and associated devices at the same or different sites. *See also* local area network, wide area network.

In television broadcasting, a chain of radio or television broadcasting stations linked by wire or microwave relay, or a company that produces the programs for these stations.

**Network Driver Interface Specification**
(NDIS) In Microsoft® Windows® networking, the Microsoft/3Com specification for the interface between device drivers and a network. All transports call the NDIS interface to access and work with network interface cards (NICs). Using NDIS, developers can write hardware device drivers that are independent of a target operating system. Broadcast Architecture supports NDIS version 5.0.

**network interface card**
(NIC) An printed circuit board, adapter or other device used to connect a computer to a network.

**NIC**
*See* network interface card.

**NIC miniport**
Miniport for the Broadcast Architecture network interface card (NIC). The NIC miniport supports Network Driver Interface Specification (NDIS).

**NIC miniport driver**
*See* NIC miniport.

**North American Basic Teletext Specification**
(NABTS) An open standard for transmission of data over the television vertical blanking interval (VBI). NABTS is an established standard for television data transmission, and it is in use by broadcasters and systems integrators in North America, Europe, South America, and the Far East.

**NTSC**
(National Television System Committee standard) Standard regulating analog television signals in North America, Japan, and parts of South America, originated by the National Television System Committee. North America and Japan established NTSC color standards to make signals compatible with black and white transmissions. NTSC is based on the 60-hertz rate of U.S. electrical mains. An NTSC set can display 525 scan lines at approximately 30 frames per second, but nonpicture lines and interlaced scanning methods make for an effective resolution limit of about 340 lines. NTSC bandwidth is 4.2 megahertz. *See also* PAL, SECAM.

# O

[This is preliminary documentation and subject to change.]

**object**
A computer programming term describing a software component that contains data or functions accessed through one or more defined interfaces. In Java and C++, an object is an instance of a object class.

**octet**
A group of eight bits capable of representing 256 different values. Internet Protocol (IP) addresses are typically represented in dotted-decimal notation — that is, with the decimal values of each octet of the address separated by periods, for example 138.57.7.27.

**operating system**
Software responsible for controlling the allocation and usage of computer hardware resources

such as memory, CPU time, disk space, and peripheral devices.

**opportunistic bandwidth**

Bandwidth granted whenever possible during the requested period, as opposed to guaranteed bandwidth, which is actually reserved for a given transmission.

**output driver**

A dynamic-link library that a Microsoft Multicast Router calls to package and transmit a stream in the format required by a particular broadcast output device. An output driver is valid only for the hardware and software configuration for which it was created, and thus each type of broadcast output device requires a different output driver.

## P

[This is preliminary documentation and subject to change.]

**packet**

A unit of information transmitted as a whole from one device to another on a network. In packet-switching networks, a packet is defined more specifically as a transmission unit of fixed maximum size that consists of binary digits representing both data and a header containing an identification number, source and destination addresses, and sometimes error-control data.

**PAL**

(Phase Alternation by Line standard) The analog television standard for much of Europe, excepting France, Russia, and most of Eastern Europe, which use SECAM. As with SECAM, PAL is based on a 50-hertz power rate, but it uses a different encoding process. It displays 625 scan lines and 25 frames per second and offers slightly better resolution than the NTSC standard used in the North America and Japan. PAL bandwidth is 5.5 megahertz.

**pay per view**

(PPV) A revenue-enhancing system in which cable or satellite customers are charged for watching a single movie or event. This system contrasts with premium cable services, which are paid for monthly regardless of usage.

**PC 97**

The system and peripheral design elements required for a computer to bear the 1997–1998 "Designed for Microsoft® Windows®" logo. *PC 97 Hardware Design Guide* defines these requirements.

**PC 98**

The system and peripheral design elements required for a computer to bear the 1998–1999 "Designed for Microsoft® Windows®" logo. *PC 98 Checklist*, an addendum to *PC 97 Hardware Design Guide*, defines these requirements.

**pin**

In the Component Object Model (COM), an object that represents the point where a unidirectional data stream connects to a filter. An input pin receives media samples, and an output pin passes media samples to the next filter.

**Plug and Play**

A design philosophy and set of specifications that describe changes to hardware and software for the personal computer and its peripherals. These changes make it possible to automatically identify and arbitrate resource requirements among all devices and buses on a computer. Plug and Play specifies a set of application programming interface (API) elements that are used in addition to existing driver architectures.

**port**

Generally, the address at which a device such as a network interface card (NIC), serial adapter, or parallel adapter communicates with a computer. Data passes in and out of such a port. In Internet Protocol, however, port signifies an arbitrary value used by the Transmission Control Protocol/Internet Protocol (TCP/IP) and User Datagram Protocol/Internet Protocol (UDP/IP) to supplement an IP address so as to distinguish between different applications or protocols residing at that address. Taken together, an IP address and a port uniquely identify a sending or receiving application or process. *See also* port driver.

**port driver**

An interface library of functions and services provided to a minidriver so that the minidriver can communicate with the operating system. The MPEG port driver is one example.

**PPV**

*See* pay per view.

**Program Guide**

An on-screen guide that lists television and cable programs and broadcast time slots. This Broadcast Architecture component receives current information about program schedules and maintains these in the Guide database. The Program Guide enables the viewer to review, select, and control what programs will be watched.

**push model**

A broadcast model in which a server sends information to a large number of clients on its own schedule, without waiting for requests. The clients scan the incoming information, save the parts they have been instructed to save, and discard the rest. Because the push model eliminates the need for requests, it allows one server to handle as much data as it has bandwidth to send. The push model contrasts with the pull model, in which each client requests information from a server. The pull model is more efficient for interactively selecting specific data to receive but uses excessive bandwidth when many clients request the same information.

## Q

[This is preliminary documentation and subject to change.]

**query**

A request that specific data be retrieved, modified, or deleted.

## R

[This is preliminary documentation and subject to change.]

**radio frequency**

(RF) The portion of the electromagnetic spectrum with frequencies between 10 kilohertz and 300 gigahertz. This portion includes the frequencies used for radio and television transmission.

**RAM**

Random access memory. RAM is semiconductor-based memory within a personal computer or other hardware device that can be rapidly read and written to by a computer's microprocessor or other devices. It does not generally retain information when the computer is turned off. *See also* ROM.

**rating**

A category applied to movies and other programs to reflect the age appropriateness of their content, particularly to help parents decide whether children should view the shows. Possible Motion Picture Association of America (MPAA) ratings include NR (not rated), G (general), PG (parental guidance suggested), PG-13 (parents strongly cautioned), R (restricted), NC-17 (no children under 17).

**registry**

In the Microsoft® Windows® 98 and Microsoft® Windows NT® operating systems, a hierarchical database that provides a repository for information about a computer's hardware and software configuration.

**renderer filter**

A filter that renders the contents of a stream on an output device such as a computer monitor, sound card, or printer. Renderer filters have only input pins.

**resource**

A piece of static data, such as a dialog box, that can be used by more than one application or in more than one place within an application. Alternatively, any part of a computer or network, such as a disk drive, printer, or memory, that can be used by a program or process.

**RF**

*See* radio frequency.

**ring 0**

The operating system kernel or core—in other words, the portion that allocates system resources; manages memory, files, and peripheral devices; maintains the time and date; and starts applications. *See also* kernel mode.

**ring 3**

The application layer. Ring 3 includes applications, dynamic-link libraries (DLLs), and DirectShow filters.

**ROM**

Read-only memory. ROM is semiconductor-based memory within a personal computer or other hardware device that contains instructions or data that can be read but not modified. *See also* RAM.

**router**

A device that helps local area networks (LANs) and wide area networks connect and interoperate. A router can connect LANs that have different network topologies, such as Ethernet and token ring. Routers choose the best path for a packet, optimizing network performance.

## S

[This is preliminary documentation and subject to change.]

**satellite receiver card**
A broadcast receiver card that receives and processes data sent by satellite.

**satellite uplink**
The system that transports a signal to a satellite for broadcast. Signals usually come to the uplink through multiplexers.

**SECAM**
(Sequential Couleur á Memoire, or Sequential Color with Memory) The television standard for France, Russia, and most of Eastern Europe. As with PAL, SECAM is based on a 50-hertz power rate, but it uses a different encoding process. Devised earlier than PAL, its standards reflect earlier technical limitations. *See also* NTSC.

**Secure Electronic Transaction**
(SET) The Microsoft standard for safely transmitting Visa or Mastercard information to support electronic commerce over the Internet.

**server**
On a network, a computer running software that provides data and services to clients over the network. The term server can also apply to a software process, such as an Automation server, that similarly sends information to clients and that appears on the same computer as a client process, or even within the same application.

**server lookup**
The process by which a content server application finds for a specified data service the names of the appropriate Microsoft Multicast Router, address reservation server, and bandwidth reservation server, along with the Internet Protocol (IP) multicast addresses reserved for global and local announcements for that data service. Because there are likely to be backup machines for the servers and the router, these names can each resolve to more than one actual IP address. For more information, see the MSBDN Services Overview section of the Broadcast Architecture Programmer's Reference. *See also* IP multicast address assignment.

**service provider**
In Broadcast Architecture, a business that provides a broadcast data service.

**Session Announcement Protocol**
(SAP) An Internet protocol for announcements that defines a header in binary format that precedes the Session Description Protocol (SDP) portion of an announcement. A draft document about SAP has been produced by the Internet Engineering Task Force (IETF).

**Session Description Protocol**
(SDP) A textual Internet protocol for the announcements intended to announce and initiate multimedia sessions. Currently a work in progress, SDP is defined in a draft document produced by the Internet Engineering Task Force (IETF). *See also* Session Announcement Protocol.

**session key**
A Triple DES key used to encrypt broadcast data content.

**SET**
*See* Secure Electronic Transaction.

**set-top box**
In standard cable or satellite systems, this device converts and decodes the incoming signal into a form that can be received by a standard television set. The device usually sits on top of the viewer's television.

**show reference**
A data format that specifies different information about television shows, such as the time the

show is broadcast, the channel the show is broadcast on, and the show's duration. Show references allow software based on Broadcast Architecture to exchange references to television shows.

**show reminder**

Tasks in Task Scheduler that remind a user to watch or record a television broadcast. Show reminders can be set by TV Viewer and by applications developed by independent software vendors.

**signal-to-noise ratio**

(S/N) The amount of power by which a signal exceeds the amount of channel noise at the same point in transmission. This amount is measured in decibels and indicates the clarity or accuracy with which communication can occur.

**smart card**

A circuit board the size of a credit card with built-in logic, memory, or firmware that gives it storage and decision-making ability, generally for purposes of purchasing, funds transfer, or identification and validation.

**S/N**

*See* signal-to-noise ratio.

**source filter**

A filter that receives data from a source, such as an MPEG file, and introduces it into a filter graph. Source filters have one or more output pins.

**spin lock**

A data type that provides a synchronization mechanism for protecting resources shared by kernel-mode threads. A thread acquires a spin lock before accessing protected resources. The spin lock keeps any thread but the one holding the spin lock from using the resource. A thread waiting for the spin lock loops, attempting to acquire the spin lock, until it is released by the thread that holds the lock.

**station**

An establishment equipped for radio or television transmission.

**stream**

A collection of data sent over a data channel in a sequential fashion. The bytes are typically sent in small packets, which are reassembled into a contiguous stream of data. Alternatively, the process of sending such small packets of data.

**streaming architecture**

A model for interconnection of stream-processing components, in which applications dynamically load data as they output it. Dynamic loading means data can be broadcast continuously. *See also* WDM streaming.

**streaming data**

Data continuously broadcast to an application. For example, a broadcast client's user might subscribe to continuously broadcast sports scores.

**string**

Data composed of a sequence of characters, usually representing human-readable text.

**stub network**

A network that carries packets only to and from local devices. Even if a stub network has paths to more than one network, it does not carry traffic for other networks.

**subgenre**

In Broadcast Architecture, a subset within a genre, for example science fiction, Western, or soap opera.

**super VGA**

(SVGA) A video standard established by the Video Electronics Standards Association (VESA) to provide high-resolution color display on IBM-compatible computers. SVGA supports the Video Graphics Array (VGA) standard.

**SVGA**

*See* super VGA.

**S-video**

Video in which chrominance and luminance signals are sent separately. This separate transmission produces a sharper video image than composite video, because it has no color artifacts. For example, S-video has no traveling dots or shimmering along color-change lines.

## T

[This is preliminary documentation and subject to change.]

**Task Scheduler**

A scheduling service and user interface that is available as a common resource with the Microsoft® Windows® 98 operating system. Windows 98 includes Task Scheduler as a Component Object Model (COM) interface. Task Scheduler manages all aspects of job scheduling: starting jobs, enumerating currently running jobs, tracking job status, and so on. In Task Scheduler, the user works with a folder under My Computer called Scheduled Tasks, which lists currently scheduled items and enables creation of new tasks. The Task Scheduler replaces the Microsoft® Windows® 95 System Agent.

**TCP/IP**

*See* Transmission Control Protocol/Internet Protocol.

**Television System Services**

(TSS) An Automation interface designed to enable developers to write Windows 98 – based applications that use broadcast television. TSS provides a user model with password security and interapplication scheduling of television shows for viewing or recording. TSS is designed to be used with the Guide database, the Video control or its underlying Microsoft® DirectShow™ filters, the Video Access server, and Guide database loaders. *See also* Broadcast Architecture subsystems.

**theme**

A category to which individual television programs are assigned within the Guide database. A theme allows a program episode to be associated with multiple genre/subgenre pairs.

**thread**

One of several paths of execution in a program or process in which all paths can execute in parallel. Each thread can perform a different function, or many threads can cooperate to perform a larger task.

**time to live**

(TTL) A value in the range 0 through 255 that defines the scope within which multicast packets should be sent over a network using Internet Protocol (IP). The scope is defined in terms of how local or remote a packet's destination is. Each router decrements the TTL by one. When the value reaches a predefined lower limit, the router throws the packet away. Current multicast backbone (MBONE) requirements, available at the ftp://ftp.isi.edu/mbone/faq.txt site, define the following standard scopes: local network, 1; local site, 15; region, 63; world, 127. Other settings may have local meaning; for example, 31 might indicate all sites within a particular organization.

**transform filter**

A filter that modifies the contents of a stream. Transform filters usually have both input and output pins.

**Transmission Control Protocol/Internet Protocol**

(TCP/IP) A networking protocol that provides reliable communications across interconnected networks made up of computers with diverse hardware architectures and operating systems. The TCP portion of the protocol, a layer above IP, is used to send a reliable, continuous stream of data and includes standards for automatically requesting missing data, reordering IP packets that might have arrived out of order, converting IP datagrams to a streaming protocol, and routing data within a computer to make sure the data gets to the correct application. The IP portion of the protocol includes standards for how computers communicate and conventions for connecting networks and routing traffic. *See also* User Datagram Protocol/Internet Protocol (UDP/IP).

**transponder**

A device on a satellite that receives and amplifies uplink signals in radio frequencies that come from an earth station. Once the signal is received, the transponder redirects it back toward its destination earth station within the particular satellite's footprint. Each transponder is responsible for a small frequency range, and a satellite is typically made up of 10 to 40 transponders. Unlike in traditional broadcast television, each transponder has enough bandwidth to carry multiple television channels, as well as data. Any one transponder might carry anywhere from 10 low resolution television channels to 2 high-definition television (HDTV) channels, or some combination. *See also* multiplexer.

**transport layer**

The fourth of the seven layers in the International Organization for Standardization's Open Systems Interconnection (OSI) model for standardizing computer-to-computer communications. The transport layer is one level above the network layer and is responsible for error detection and correction, among other tasks. Error correction ensures that the bits delivered to the receiver are the same as the bits transmitted by the sender, in the same order and without modification, loss, or duplication. The transport layer is the highest of the three layers (data link, network, and transport) that help move information from one device to another. *See also* data link layer.

**Triple DES**

A method of encryption in which the Data Encryption Standard (DES) algorithm is applied three times in a row using three different session keys. These three keys are combined into one Triple DES session key. Triple DES encryption is much more resilient to differential and plaintext attacks than DES.

**TSS**

*See* Television System Services.

**tuner**

An electronic component that locks onto a selected channel and filters signals such as audio and video from that frequency for amplification and display.

**tuning space**

A set of nonoverlapping channels that are all available through the same type of physical channel tuner, such as a broadcast receiver card or analog cable tuner. A broadcast client with multiple tuning devices can provide channels from multiple tuning spaces.

**TV Viewer**

A Broadcast Architecture component that enables users to select television channels and that displays enhanced television programs. TV Viewer uses an ActiveX control to display conventional television shows, and it hosts a Hypertext Markup Language (HTML) browser to display enhanced content for those shows at the same time.

**twisted-pair cable**

A communications medium consisting of two thin, insulated wires, generally made of copper, that are twisted together. Standard telephone connections are often referred to as "twisted pair."

## U

[This is preliminary documentation and subject to change.]

**unicast**

A point-to-point networking model in which a packet is duplicated for each address that needs to receive it. *See also* multicast.

**upstream**

One-way data flow from broadcast client to head end. *See also* downstream.

**User Datagram Protocol/Internet Protocol**

(UDP/IP) A networking protocol used to send large unidirectional packets across interconnected networks made up of computers with diverse hardware architectures and operating systems. The UDP portion of the protocol, a networking layer above IP, is used to send unidirectional packets of up to 64 kilobytes in size and includes standards for routing data within a single computer so it reaches the correct client application. The IP portion of the protocol includes standards for how computers communicate and conventions for connecting networks and routing traffic. *See also* Transmission Control Protocol/Internet Protocol.

**user mode**

Software processing that occurs at the application layer, ring 3.

**utility filter**

A filter that supports the processing of audio and video without necessarily directly manipulating the broadcast stream. *See also* renderer filter, source filter, and transform filter.

## V

[This is preliminary documentation and subject to change.]

**VBI**

*See* vertical blanking interval.

**vertical blanking interval**

(VBI) The time period in which a television signal is not visible on the screen because of the vertical retrace (that is, the repositioning to top of screen to start a new scan). Data services can be transmitted using a portion of this signal. In a standard NTSC signal, perhaps 10 scan lines are potentially available per channel during the VBI. Each scan line represents a data transmission capacity of about 9600 baud.

**VGA**

*See* Video Graphics Array.

**Video Access server**

An out-of-process server that handles device contention among multiple instances of the TV Viewer ActiveX control. Device contention is conflict over which process controls a particular device. The Video Access server also checks viewer permissions when viewers tune to television programs.

**Video control**

A Broadcast Architecture component that applications use to control how the Video Access server displays audio and video streams.

**Video Graphics Array**

(VGA) A video standard created by IBM that supports several modes, including a graphics mode of 640 horizontal pixels by 480 vertical pixels with 2 or 16 simultaneous colors and a graphics mode of 320 horizontal pixels by 200 vertical pixels with 256 colors. The term VGA chip is often used generically to refer to a video controller chip.

**virtual device driver**

(VxD) A 32-bit, protected-mode driver that manages a system resource, such as a hardware device or installed software, so that more than one application can use the resource at the same time. V*x*D refers to a general virtual device driver — the *x* represents the type of device driver. For example, a virtual device driver for a display device is known as a VDD, a virtual device driver for a timer device is a VTD, a virtual device driver for a printer device is a VPD, and so on.

**VxD**

*See* virtual device driver.

# W

[This is preliminary documentation and subject to change.]

**WDM**

*See* Windows Driver Model.

**wide area network**

(WAN) A communications network that connects geographically separated areas. *See also* local area network.

**WDM streaming**

An extension of the Microsoft® DirectShow™ application programming interface (API) based on the Windows Driver Model (WDM). WDM streaming provides the kernel connection and streaming services used by the WDM stream class driver and by components of Microsoft® Windows NT® and Microsoft Windows® 98. In these operating systems, WDM streaming provides low-level services in Ring 0 for the lowest latency streaming. DirectShow provides higher-level features and control.

**Windows Driver Model**

(WDM) A Microsoft 32-bit driver model based on the driver model within the Microsoft® Windows NT® operating system. WDM provides a common architecture of input/output (I/O) services and device drivers for specific driver classes. Both Windows NT and the Microsoft®

Windows® 98 operating system support WDM.

**Windows Sockets**

(WinSock) An application programming interface that provides access to multiple transport protocols, including Transmission Control Protocol/Internet Protocol (TCP/IP), the Internet standard. WinSock provides a network abstraction layer that allows applications to receive and send network data without requiring information about the network involved. WinSock also supports networking capabilities such as real-time multimedia communications.

**World Wide Web**

(the Web) A hypertext-based, distributed information system created in Switzerland and used for exploring the Internet. Users may create, edit, or browse hypertext documents on the Web.

**wrapper**

A function that provides an interface to another function.

## Y

[This is preliminary documentation and subject to change.]

**YUV**

True-color encoding that uses one luminance value (Y) and two chroma values (UV).

## README.TXT

```
For information on system requirements for Broadcast Architecture, see the Client H

NOTE: For the Broadcast Architecture samples to have maximum value, you should have
      following components installed:

   1. Microsoft  Windows  98, including the optional TV Viewer component.

   2. Microsoft Visual Basic version 5.0. For more information on installing
      and using Visual Basic 5.0, see the Visual Basic 5.0 documentation.

   3. Microsoft Visual C++ version 5.0. For more information on installing
      and using Visual C++ 5.0, including the Microsoft Foundation Class
      (MFC) Library, see the Visual C++ 5.0 documentation.
```

## README.TXT

```
Loader DLL to demonstrate loading data into the Guide database


SUMMARY
=======

Load is a sample loader that shows how to create a loader that edits data for the G

MORE INFORMATION
================

The following information describes the Load sample.

Requirements
-----------------------------
Load.exe must be compiled using Microsoft Visual C++ 5.0 with Microsoft Foundation 

You must configure your environment before compiling Load.exe. First, make sure tha

To Compile Load
-----------------------------

Use the following command to compile the sample:

   nmake load.mak

For make options, see the makefile header comments.

If the build tools cannot find Sdkutil.lib, use the Samples\Com\Common sample to bu

To Run Load
-----------------------------

Load runs under the Broadcast Architecture program Loadstub.exe. The first step is 
```

```
Here is an example of a registry entry:

HKEY_LOCAL_MACHINE\Software\Microsoft\TV Services\Guide\Loaders\{1234 9ABC 9998 DDD
Namec:\Windows\Program Files\TV Viewer\Load.dll

Now you can run Loadstub with the GUID in the /L: command line parameter. Here is t

loadstub /L:{1234 9ABC 9998 DDD}

Load Files
-------------
Load.cpp is the main program file for the loader. It contains the entry function ca
```

# CMDPROC.H

```
/////////////////////////////////////////////////////////////////////////////
// cmdproc.h
// Copyright (C) 1996 Microsoft Corp.
//
//  more flexible replacement for mfc CCommandLineInfo

/////////////////////////////////////////////////////////////////////////////
// CCommandLineInfo

#ifndef CMDPROC_H
#define CMDPROC_H


class CCommandLineProc : public CObject
{
public:
    // process the command line for switch based arguments
    BOOLEAN ProcessCommandLine(int iSC, int &argc, _TCHAR **argv);
protected:
    typedef void (CCommandLineProc::*PMFNCmdProc)(CString &csArg);
    class CArgProcTable {
    public:
        int m_iIDS;  // string resource of command switch
        PMFNCmdProc m_Cmd;    // argument processing function
    };
    friend CArgProcTable;
    static CArgProcTable acapArgs[];
    // remove any desired positional arguments
    virtual BOOLEAN GetPositionalArgs(int &argc, _TCHAR **argv);

    // this function deletes the argument at iPos by copy the remaining
    // elements of argv 1 to the left
    inline void CCommandLineProc::DeleteArg(int iPos, int &argc, _TCHAR **argv)
    {
        for (int k = iPos + 1; k < argc; k++) {
        argv[k - 1] = argv[k];
        }
        argc--;
    }
};


#endif
// end of file - cmdproc.h
```

## CMDPROC2.CPP

```
// cmdproc.cpp : replacement for mfc CCommandLineInfo
//
//
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// See the reference for detailed information regarding
// Broadcast Architecture.

#include "stdafx.h"
#include "cmdproc.h"

BOOLEAN
CCommandLineProc::ProcessCommandLine(int iSC, int &argc, _TCHAR **argv)
{
    CString csSwitchChars;
    csSwitchChars.LoadString(iSC);
    CString csParam;

    if(acapArgs[0].m_Cmd == NULL)
{
        return TRUE;
    }
  int i = 1;
    while(i < argc)
{
        csParam = argv[i];
if(!_tcscspn(csParam, csSwitchChars))
{
            // its a switch
// remove flag specifier
csParam = csParam.Right(csParam.GetLength() - 1);
            // find the function for this switch
            CString csSwitch, csArg;

            for(int j = 0; acapArgs[j].m_Cmd != NULL; j++)
{
                csSwitch.LoadString(acapArgs[j].m_iIDS);
                if(!_tcsnccmp(csParam, csSwitch, csSwitch.GetLength()))
{
                    break;
                }
            }
            if(acapArgs[j].m_Cmd != NULL)
{
```

```
                csArg = csParam.Right(csParam.GetLength() - csSwitch.GetLength());
                (this->*(acapArgs[j].m_Cmd))(csArg);
                DeleteArg(i, argc, argv);
            }
else
{
                i++;  // skip this one
            }
}
else
{
            // its positional so skip it.
            i++;
        }

}
    return GetPositionalArgs(argc, argv);
}

BOOLEAN
CCommandLineProc::GetPositionalArgs(int &argc, _TCHAR **argv)
{
    // default implementation provides no positional args
    return TRUE;
}
```

## LOAD.CPP

```
// load.cpp : Guide database loader sample program
//
//
// This is a part of the Microsoft Foundation Classes C++ library.
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// See the reference for detailed information regarding
// Broadcast Architecture.

#include "stdafx.h"
#include "Load.h"
#include "tssutil.h"


/////////////////////////////////////////////////////////////////////////////
// CLoadApp

BEGIN_MESSAGE_MAP(CLoadApp, CWinApp)
//{{AFX_MSG_MAP(CLoadApp)
// NOTE - the ClassWizard will add and remove mapping macros here.
//    DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
```

```
// CLoadApp construction

CLoadApp::CLoadApp()
{
// TODO: add construction code here,
// Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CLoadApp object

CLoadApp theApp;


//
// This is the external entry point defined in the .DEF file that is used by
// the generic loader app to call into the Sample loader.
//
extern "C"
{
ExitCodeList APIENTRY
EPG_DBLoad(int &argc, _TCHAR **argv, CdbDBEngine &db, PFNFORCEQUIT pfnForceQuit)
{

    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    ExitCodeList rc = theApp.EPG_DBLoad(argc, argv, db, pfnForceQuit);

if(theApp.m_lpfnDaoTerm)
(*theApp.m_lpfnDaoTerm)();

return rc;
}
};


CLoadApp::CLoadCommandLineProc::CLoadCommandLineProc(void)
{
    m_fPartial = FALSE;
m_csRead = "";
}


void
CLoadApp::CLoadCommandLineProc::Partial(CString &csArg)
// Process the cmd line switch that allows us to be
// told to do a partial update
{
    m_fPartial = TRUE;
    return;
}


void
CLoadApp::CLoadCommandLineProc::Read(CString &csArg)
// Process the cmd line switch that gives a location to read loader data from
{
    m_csRead = csArg;
    return;
}


void
```

```
CLoadApp::CLoadCommandLineProc::Help(CString &csArg)
// Display the command line usage
{
    AfxMessageBox(IDS_USAGE, MB_OK | MB_ICONSTOP);
    return;
}

// This table tells the command line processor what to do for each valid argument.
CLoadApp::CLoadCommandLineProc::CArgProcTable
CLoadApp::CLoadCommandLineProc::acapArgs[] =
{
    IDS_SWPARTIAL, (void (CCommandLineProc::*)(class CString &))CLoadApp::CLoadComm
    IDS_SWREAD, (void (CCommandLineProc::*)(class CString &))CLoadApp::CLoadCommand
    IDS_SWHELP, (void (CCommandLineProc::*)(class CString &))CLoadApp::CLoadCommand
    -1, NULL,
};


BOOLEAN
CLoadApp::CLoadCommandLineProc::GetPositionalArgs(int &argc, _TCHAR **argv)
{
    return TRUE;
}


// This is the main entry point into the loader class.
ExitCodeList
CLoadApp::EPG_DBLoad(int &argc, _TCHAR **argv, CdbDBEngine &db
, PFNFORCEQUIT pfnForceQuit)
{
ExitCodeListrc;

    if(!m_clpCmds.ProcessCommandLine(IDS_SWITCHCHARS, argc, argv))
return EXIT_USAGE;

theApp.m_pDAODB = &db;
theApp.m_pfnForceQuit = pfnForceQuit;

    try
    {
        if(EXIT_OK != (rc = InitMembers()))
        return (rc);

        rc = ProcessInput(m_clpCmds.m_fPartial, db);

        return (rc);
    }
    catch (CException *e)
    {
        e->Delete();

        return (EXIT_FAIL_CEXCEPTION);
    }
    catch (CdbException cdbe)
    {
return (EXIT_FAIL_DBEXCEPTION);
    }
catch (ExitCodeList rc)
    {
return (rc);
    }
    catch (...)
    {
```

```
return (EXIT_FAIL);
    }
}

// This reinitializes certain app class members
ExitCodeList
CLoadApp::InitMembers(void)
{
DWORD rc;

    // Incoming data is all gmt.  Set up app class time info
    // for conversions
    TIME_ZONE_INFORMATION tzi;
    DWORD tzrc;
    tzrc = ::GetTimeZoneInformation(&tzi);

    CTimeSpan bias(0, 0, tzi.Bias, 0);
    switch(tzrc)
{
    case TIME_ZONE_ID_UNKNOWN:
    break;
case TIME_ZONE_ID_DAYLIGHT:
{
CTimeSpan temp(0, 0, tzi.DaylightBias, 0);
bias += temp;
        break;
}
case TIME_ZONE_ID_STANDARD:
{
            CTimeSpan temp(0, 0, tzi.StandardBias, 0);
            bias += temp;
break;
}
    default:
THROWASSERT(0, EXIT_FAIL_GETTIMEZONE);
        break;
}
    m_odtsTimeZoneAdjust.SetDateTimeSpan(bias.GetDays(), bias.GetHours(), bias.GetM
m_codtGuideStartTime.SetStatus(COleDateTime::invalid);
m_codtGuideEndTime.SetStatus(COleDateTime::invalid);

    DWORD dwBytes = sizeof(m_lTuningSpace);
    rc = TSS_GetTuningIDs(SZDTVLOADGUID, (DWORD *)&m_lTuningSpace, dwBytes);
    if((!rc) /*|| (dwBytes != sizeof(m_lTuningSpace)) GFS - How cound this be true?
        return EXIT_FAIL_GETTUNINGSPACE;

m_covTuningSpace = m_lTuningSpace;

    //aux
    rs = NULL;
    r = NULL;
    bp = NULL;
    g = NULL;
    sg = NULL;
    sr = NULL;
    // main
    n = NULL;
    s = NULL;
    c = NULL;
    cp = NULL;
    csr = NULL;
    e = NULL;
    ep = NULL;
    ts = NULL;
    t = NULL;

    return EXIT_OK;
}


ExitCodeList CLoadApp::ProcessInput(BOOLEAN fPartialUpdate, CdbDBEngine &db)
{
    ExitCodeList rc = EXIT_OK;

    COleDateTime m_odtStart(COleDateTime::GetCurrentTime());
    m_covNow = m_odtStart + m_odtsTimeZoneAdjust;

CString csWSP;
csWSP.LoadString(IDS_LOADERWORKSPACE);

    try
    {
OpenTables();

db[csWSP].BeginTrans();

Handle(fPartialUpdate);

        ClearOldEntries(db);  // delete anything from mpg time span that wasn't ref

ClearDanglingRefs(db);  // cleanup unused auxiliary records

BlockCommit(db, EPGLDR_ACTIVE_COMMIT_STARTING, EPGLDR_ACTIVE_COMMIT_ENDING);

CloseTables();
}
    catch (CException *e)
    {
        e->Delete();

db[csWSP].Rollback();

        return (EXIT_FAIL_CEXCEPTION); // failed
    }
    catch (CdbException cdbe)
    {
db[csWSP].Rollback();

return (EXIT_FAIL_DBEXCEPTION);
    }
catch (ExitCodeList rc)
    {
db[csWSP].Rollback();

return (rc);
    }
catch (...)
    {
db[csWSP].Rollback();

return (EXIT_FAIL);
    }

#ifdef _DEBUG
    if (afxTraceFlags & traceDatabase)
    {
        COleDateTime end_parse(COleDateTime::GetCurrentTime());
        COleDateTimeSpan duration(end_parse - m_odtStart);
        afxDump << "end parse. length = " << ((ULONG)duration.GetTotalSeconds()) <<
    }
#endif

    return rc;
}

// execute deletes on each table in proper order for
// all records whose last tx hasn't been updated and
// who's attached to stuff that overlaps in the mpg's time frame
void CLoadApp::ClearOldEntries(CdbDBEngine &db)
{
CString csWSP;
csWSP.LoadString(IDS_LOADERWORKSPACE);

    try
    {
CdbDatabase database = db[csWSP][(LONG) 0];
db[csWSP].BeginTrans();

COleDateTime codtStartTimeMinusDay = m_codtGuideStartTime;
COleDateTimeSpan codtsDelta;
codtsDelta.SetDateTimeSpan(1, 0, 0, 0);

codtStartTimeMinusDay -= codtsDelta;

// delete old timeslots with end time < mpg start time
// this removes any showings that are finished
        ExecuteActionQuery(database, IDS_DELETE_EXPIRED_TS, & (COleVariant) m_lTuni

        // delete unupdated timeslots with last update < now and start time < guide
        // this removes any showings that were in the mpg last time the loader ran
        // but that aren't there now
        ExecuteActionQuery(database, IDS_DELETE_OMITTED_TS, & (COleVariant) m_lTuni
&m_covNow, & (COleVariant) m_codtGuideStartTime, & (COleVariant) m_codtGuideEndTime

        // the previous actions may have left episode or channels records which are
        // by any time slots.

Commit(db);

#ifdef _DEBUG
        if (afxTraceFlags & traceDatabase)
            TRACE0("clear old entries committed\r\n");
#endif
    }
    catch (CException *e)
    {
        e->Delete();

db[csWSP].Rollback();

#ifdef _DEBUG
afxDump << "ClearOldEntries CException catch handler\r\n";
#endif
        throw e;
    }
    catch (...)
    {
db[csWSP].Rollback();
#ifdef _DEBUG
afxDump << "ClearOldEntries ... catch handler\r\n";
#endif
        throw (EXIT_FAIL);
    }
}


// execute deletes on each table in proper order for
// all auxiliary record which aren't used by anyone
// we now delete these dangling records and this completes the differencing portion
// of the partial update
void CLoadApp::ClearDanglingRefs(CdbDBEngine &db)
{
CString csWSP;
csWSP.LoadString(IDS_LOADERWORKSPACE);

try
{
CdbDatabase database = db[csWSP][(LONG) 0];
db[csWSP].BeginTrans();

        ExecuteActionQuery(database, IDS_DELETE_DANGLING_C);
        ExecuteActionQuery(database, IDS_DELETE_DANGLING_E);
        ExecuteActionQuery(database, IDS_DELETE_DANGLING_S);
ExecuteActionQuery(database, IDS_DELETE_DANGLING_THEME);

Commit(db);

#ifdef _DEBUG
        if (afxTraceFlags & traceDatabase) {
            TRACE0("dangling deletes committed\r\n");
        }
#endif
    }
    catch (CException *e)
    {
        e->Delete();

db[csWSP].Rollback();

#ifdef _DEBUG
afxDump << "ClearDanglingRefs CException catch handler\r\n";
#endif
        throw e;
    }
catch (...)
{
db[csWSP].Rollback();

#ifdef _DEBUG
afxDump << "ClearDanglingRefs ... catch handler\r\n";
#endif
        throw (EXIT_FAIL);
    }
}

// This method executes a query in the database
void
CLoadApp::ExecuteActionQuery(CdbDatabase &qd, int iStringID, COleVariant *p0
, COleVariant *p1, COleVariant *p2, COleVariant *p3)
{
    CString csDelQuery;
```

```
  if ((*m_pfnForceQuit)())
throw (EXIT_ABORT);

csDelQuery.LoadString(iStringID);

CdbQueryDef qdbdd;

qdbdd = qd.QueryDefs.Item((LPCTSTR) csDelQuery);
    if (p0 != NULL)
        qdbdd.Parameters[(LONG) 0].SetValue(*p0);
    if (p1 != NULL)
        qdbdd.Parameters[(LONG) 1].SetValue(*p1);
    if (p2 != NULL)
        qdbdd.Parameters[(LONG) 2].SetValue(*p2);
    if (p3 != NULL)
        qdbdd.Parameters[(LONG) 3].SetValue(*p3);

    qdbdd.Execute();

    return;
}


void
CLoadApp::Commit(CdbDBEngine &db, LONG lStartMessage, LONG lEndMessage)
{
UINT uiMessage = RegisterWindowMessage(SZLOADERSTUBGUID);

if(0 != lStartMessage)
::PostMessage(HWND_BROADCAST, uiMessage, lStartMessage, 0);

CString csWSP;
csWSP.LoadString(IDS_LOADERWORKSPACE);
db.Idle();
db[csWSP].CommitTrans();

if(0 != lEndMessage)
::PostMessage(HWND_BROADCAST, uiMessage, lEndMessage, 0);
}


void
CLoadApp::BlockCommit(CdbDBEngine &db, LONG lStartMessage, LONG lEndMessage)
{
UINT uiMessage = RegisterWindowMessage(SZLOADERSTUBGUID);

if(0 != lStartMessage)
::SendMessageTimeout(HWND_BROADCAST, uiMessage, lStartMessage, 0,
SMTO_NORMAL, 5 * 1000, NULL);
CString csWSP;
csWSP.LoadString(IDS_LOADERWORKSPACE);
db.Idle();
db[csWSP].CommitTrans();

if(0 != lEndMessage)
::PostMessage(HWND_BROADCAST, uiMessage, lEndMessage, 0);
}


// Open recordsets for each of the tables
// Set each recordset to proper index
void
```

```
CLoadApp::OpenTables(VOID)
{
     // aux
    rs = new CRatingSystemRecordset();
    rs->OpenIndexed(IDS_RS_ADDKEY, dbOpenTable, NULL, 0);

    r = new CRatingRecordset();
    r->OpenIndexed(IDS_R_ADDKEY, dbOpenTable, NULL, 0);

    bp = new CBroadcastPropertyRecordset();
    bp->OpenIndexed(IDS_BP_ADDKEY, dbOpenTable, NULL, 0);

    g = new CGenreRecordset();
    g->OpenIndexed(IDS_G_ADDKEY, dbOpenTable, NULL, 0);

    sg = new CSubGenreRecordset();
    sg->OpenIndexed(IDS_SG_ADDKEY, dbOpenTable, NULL, 0);

    sr = new CStreamTypeRecordset();
    sr->OpenIndexed(IDS_SR_ADDKEY, dbOpenTable, NULL, 0);

    // main
    n = new CNetworkRecordset();
    n->OpenIndexed(IDS_N_ADDKEY, dbOpenTable, NULL, 0);

    s = new CStationRecordset();
    s->OpenIndexed(IDS_S_ADDKEY, dbOpenTable, NULL, 0);

    c = new CChannelTRecordset();
    c->OpenIndexed(IDS_C_ADDKEY, dbOpenTable, NULL, 0);

    cp = new CChannelPropertyRecordset();
    cp->OpenIndexed(IDS_CP_ADDKEY, dbOpenTable, NULL, 0);

    csr = new CChannelStreamRecordset();
    csr->OpenIndexed(IDS_CSR_ADDKEY, dbOpenTable, NULL, 0);

    e = new CEpisodeTRecordset();
    e->OpenIndexed(IDS_E_ADDKEY, dbOpenTable, NULL, 0);

    ep = new CEpisodePropertyRecordset();
    ep->OpenIndexed(IDS_EP_ADDKEY, dbOpenTable, NULL, 0);

    ts = new CTimeSlotRecordset();
    ts->OpenIndexed(IDS_TS_ADDKEY, dbOpenTable, NULL, 0);

    t = new CThemeRecordset();
    t->OpenIndexed(IDS_T_ADDKEY, dbOpenTable, NULL, 0);
}


// close all of the tables
void
CLoadApp::CloseTables(void)
{
    //aux
    if (rs != NULL) {
        rs->CloseRecordset();
        delete rs;
        rs = NULL;
    }
    if (r != NULL) {
        r->CloseRecordset();
```

```
            delete r;
            r = NULL;
        }
        if (bp != NULL) {
            bp->CloseRecordset();
            delete bp;
            bp = NULL;
        }
        if (g != NULL) {
            g->CloseRecordset();
            delete g;
            g = NULL;
        }
        if (sg != NULL) {
            sg->CloseRecordset();
            delete sg;
            sg = NULL;
        }
        if (sr != NULL) {
            sr->CloseRecordset();
            delete sr;
            sr = NULL;
        }
        // main
        if (n != NULL) {
            n->CloseRecordset();
            delete n;
            n = NULL;
        }
        if (s != NULL) {
            s->CloseRecordset();
            delete s;
            s = NULL;
        }
        if (c != NULL) {
            c->CloseRecordset();
            delete c;
            c = NULL;
        }
        if (cp != NULL) {
            cp->CloseRecordset();
            delete cp;
            cp = NULL;
        }
        if (csr != NULL) {
            csr->CloseRecordset();
            delete csr;
            csr = NULL;
        }
        if (e != NULL) {
            e->CloseRecordset();
            delete e;
            e = NULL;
        }
        if (ep != NULL) {
            ep->CloseRecordset();
            delete ep;
            ep = NULL;
        }
        if (ts != NULL) {
            ts->CloseRecordset();
            delete ts;
            ts = NULL;
        }
        if (t != NULL) {
            t->CloseRecordset();
            delete t;
            t = NULL;
        }
    }


    // Put data into the Guide Database
    VOID
    CLoadApp::Handle(BOOL fPartialUpdate)
    {
    // Add a station
    CStationcs(AFX_RFX_LONG_PSEUDO_NULL, "WGFS", "Sample Station"
    , 0// Network ID
    , "mylogo"// Logo
    , "The Sample station");// Description
    s->UpdateRS(cs);

    // Add a channel
    COleDateTime codtDummy = COleDateTime(1999, 12, 30, 0, 0, 0);
    CChannelTcct(AFX_RFX_LONG_PSEUDO_NULL
    , -2// Tuning space
    , 241// Channel number
    , codtDummy// Start time
    , codtDummy// End time
    , 0// Length
    , cs.StationID()// Station ID
    , "The Sample Channel"// Description
    , 0// Enhancement ID
    , 0// Rating ID
    , 0// Display mask
    , 0// Payment address
    , 0// Payment token
    , COleDateTime::GetCurrentTime() + m_odtsTimeZoneAdjust); // Last update

    c->UpdateRS(cct);

    // Add an episode
    CEpisodeT cet(AFX_RFX_LONG_PSEUDO_NULL
    , "The Sample Show"// Title
    , "Bringing you samples from around the world"// Description
    , 0// Enhancement ID
    , 0// Display mask
    , 0// Theme ID
    , 0// Rating ID
    , 255// Abbreviation
    , COleDateTime::GetCurrentTime() + m_odtsTimeZoneAdjust);// last update
    e->UpdateRS(cet);

    // Put the show in a time slot
    // Start the show at the top of the next hour
    COleDateTime codtStart = COleDateTime::GetCurrentTime();
    codtStart.SetDateTime(codtStart.GetYear(), codtStart.GetMonth(), codtStart.GetDay()
    , codtStart.GetHour()+1, 0, 0);
    // Make it a half hour show
    COleDateTime codtEnd = codtStart + COleDateTimeSpan(0, 0, 30, 0);

    ts->UpdateRS(CTimeSlot(AFX_RFX_LONG_PSEUDO_NULL
    , cct.ChannelID()// Channel ID
    , cet.EpisodeID()// Eppisode ID
    , codtStart// Start time
```

```
, codtEnd// End time
, 30// Length
, 0// Payment address
, 0// Payment token
, COleDateTime::GetCurrentTime() + m_odtsTimeZoneAdjust// Last update
, FALSE// Pay-per-view
, FALSE// Closed caption
, FALSE// Stereo
, FALSE// Re-run
, FALSE// Tape inhibited
, FALSE// Other properties
, FALSE// Alternate data
, FALSE));// Alternate audio

}
```

## LOAD.H

```
// Load.h : main header file for the LOAD DLL
//

#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"// main symbols


// turn off warning about debug info truncation
#pragma warning(disable:4786)

#include <epgldrx.h>
#include <cmdproc.h>
#include "RatingSy.h"
#include "Rating.h"
#include "BrProp.h"
#include "Genre.h"
#include "SubGenre.h"
#include "StreamTy.h"
#include "ChannelT.h"
#include "ChProp.h"
#include "ChanStr.h"
#include "EpisodeT.h"
#include "EpProp.h"
#include "TimeSlot.h"
#include "Station.h"
#include "Network.h"
#include "Theme.h"


typedef unsigned char UBYTE;


#ifdef _DEBUG
#define THROWASSERT(f, errorcode)ASSERT(f)
#endif
```

```
#ifdef NDEBUG
#define THROWASSERT(f, errorcode) \
do \
{ \
if (!(f)) \
throw (errorcode); \
} while (0);
#endif


//////////////////////////////////////////////////////////////////////////////
// CLoadApp
// See Load.cpp for the implementation of this class
//

class CLoadApp : public CWinApp
{
public:
CLoadApp();

ExitCodeList EPG_DBLoad(int &argc, _TCHAR **argv, CdbDBEngine &db, PFNFORCEQUIT pfn

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CLoadApp)
//}}AFX_VIRTUAL

//{{AFX_MSG(CLoadApp)
// NOTE - the ClassWizard will add and remove member functions here.
//    DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()

protected:
    LONGm_lTuningSpace;
    COleVariant m_covTuningSpace, m_covNow;
    COleDateTimeSpan m_odtsTimeZoneAdjust;
    COleDateTimem_codtGuideStartTime, m_codtGuideEndTime;
    WORDm_wGuideLength;

CdbDBEngine *m_pDAODB;
    PFNFORCEQUIT m_pfnForceQuit;

    ExitCodeList InitMembers(void);
ExitCodeList ProcessInput(BOOLEAN fPartialUpdate, CdbDBEngine &db);

VOIDHandle(BOOL fPartialUpdate);

void Commit(CdbDBEngine &db, LONG lStartMessage = 0, LONG lEndMessage = 0);
void BlockCommit(CdbDBEngine &db, LONG lStartMessage = 0, LONG lEndMessage = 0);
    void OpenTables(VOID);
    void ClearOldEntries(CdbDBEngine &db);
    void ClearDanglingRefs(CdbDBEngine &db);
    void ExecuteActionQuery(CdbDatabase &db, int iStringID, COleVariant *p0 = NULL,
    void CloseTables();

    // aux
CRatingSystemRecordset *rs;
    CRatingRecordset *r;
    CBroadcastPropertyRecordset *bp;
    CGenreRecordset *g;
    CSubGenreRecordset *sg;
    CStreamTypeRecordset *sr;
    CThemeRecordset *t;
   // main
    CNetworkRecordset *n;
    CStationRecordset *s;
    CChannelTRecordset *c;
    CChannelPropertyRecordset *cp;
    CChannelStreamRecordset *csr;
    CEpisodeTRecordset *e;
    CEpisodePropertyRecordset *ep;
    CTimeSlotRecordset *ts;

    class CLoadCommandLineProc : public CCommandLineProc
    {
    public:
        BOOLEAN m_fPartial;
        CString m_csRead;
        CLoadCommandLineProc(void);

    protected:
        void Read(CString &csArg);
        void Partial(CString &csArg);
        void Help(CString &csArg);
        friend CCommandLineProc::CArgProcTable;
        BOOLEAN GetPositionalArgs(int &argc, _TCHAR **argv);
    } m_clpCmds;
};

/////////////////////////////////////////////////////////////////////////////
```

## CMDPROC.CPP

```
/////////////////////////////////////////////////////////////////////////////
// cmdproc.cpp
// Copyright (C) 1996 Microsoft Corp.
//
//  more flexible replacement for mfc CCommandLineInfo

/////////////////////////////////////////////////////////////////////////////
// CCommandLineProc

#include "stdafx.h"
#include "cmdproc.h"

// NOTE: this member should be used from inside a set of try/catch
// since there is a remote chance that some of the string handling
// could throw memory exceptions
BOOLEAN CCommandLineProc::ProcessCommandLine(int iSC, int &argc, _TCHAR **argv)
{
    CString csSwitchChars;
    csSwitchChars.LoadString(iSC);
    CString csParam;

    if (acapArgs[0].m_Cmd == NULL) {
        return TRUE;
    }
  int i = 1;
```

```
    while (i < argc) {
        csParam = argv[i];
if (!_tcscspn(csParam, csSwitchChars))
{
            // its a switch
// remove flag specifier
csParam = csParam.Right(csParam.GetLength() - 1);
            // find the function for this switch
            CString csSwitch, csArg;

            for (int j = 0; acapArgs[j].m_Cmd != NULL; j++) {
                csSwitch.LoadString(acapArgs[j].m_iIDS);
                if (!_tcsnccmp(csParam, csSwitch, csSwitch.GetLength())) {
                    break;
                }
            }
            if (acapArgs[j].m_Cmd != NULL) {
                csArg = csParam.Right(csParam.GetLength() - csSwitch.GetLength());
                (this->*(acapArgs[j].m_Cmd))(csArg);
                DeleteArg(i, argc, argv);
            } else {
                i++;  // skip this one
            }
} else {
            // its positional so skip it.
            i++;
        }

}
    return GetPositionalArgs(argc, argv);
}

BOOLEAN CCommandLineProc::GetPositionalArgs(int &argc, _TCHAR **argv)
{
    // default implementation provides no positional args
    return TRUE;
}
```

## RESOURCE.H

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Load.rc
//
#define IDS_SWPARTIAL                   102
#define IDS_SWHELP                      103
#define IDS_SWITCHCHARS                 104
#define IDS_USAGE                       105
#define IDS_S_ADDKEY                    135
#define IDS_BP_ADDKEY                   136
#define IDS_C_ADDKEY                    137
#define IDS_CP_ADDKEY                   138
#define IDS_CSR_ADDKEY                  139
#define IDS_E_ADDKEY                    140
#define IDS_EP_ADDKEY                   141
#define IDS_G_ADDKEY                    142
```

```
#define IDS_R_ADDKEY                    143
#define IDS_RS_ADDKEY                   144
#define IDS_SC_ADDKEY                   145
#define IDS_SR_ADDKEY                   146
#define IDS_TS_ADDKEY                   147
#define IDS_SG_ADDKEY                   148
#define IDS_N_ADDKEY                    149
#define IDS_DELETE_GENRE                201
#define IDS_DELETE_STREAM               202
#define IDS_DELETE_EPISODE              203
#define IDS_DELETE_CHANNEL              204
#define IDS_DELETE_DANGLING_BP          205
#define IDS_DELETE_DANGLING_S           206
#define IDS_DELETE_DANGLING_THEME       207
#define IDS_DELETE_DANGLING_E           208
#define IDS_DELETE_DANGLING_C           209
#define IDS_DELETE_EXPIRED_TS           210
#define IDS_DELETE_OMITTED_TS           211
#define IDS_CC                          236
#define IDS_CC_NAME                     237
#define IDS_CC_PICTOGRAM                240
#define IDS_T_ADDKEY                    245
#define IDS_MISC                        246
#define IDS_C_SEEK                      247
#define IDS_STEREO                      249
#define IDS_STEREO_NAME                 250
#define IDS_STEREO_PICTOGRAM            251
#define IDS_PPV                         252
#define IDS_PPV_NAME                    253
#define IDS_PPV_PICTOGRAM               254
#define IDS_RERUN                       255
#define IDS_RERUN_NAME                  256
#define IDS_RERUN_PICTOGRAM             257
#define IDS_RERUN_PARSE                 258
#define IDS_BLACKWHITE                  259
#define IDS_NUDITY                      260
#define IDS_VIOLENCE                    261
#define IDS_ADULT_SITUATIONS            262
#define IDS_ADULT_THEMES                263
#define IDS_ADULT_LANGUAGE              264
#define IDS_DESCRIPTION_START           265
#define IDS_DESCRIPTION_MID             266
#define IDS_DESCRIPTION_END             267
#define IDS_RATING_SYSTEM_NAME          268
#define IDS_RATING_SYSTEM_DESC          269
#define IDS_LOADERWORKSPACE             270
#define IDS_SWREAD                      271

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        129
#define _APS_NEXT_COMMAND_VALUE         32771
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

## STDAFX.CPP

```
// stdafx.cpp : source file that includes just the standard includes
//SSLoad.pch will be the pre-compiled header
//stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"
```

## STDAFX.H

```
// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
//

#define VC_EXTRALEAN// Exclude rarely-used stuff from Windows headers
#define _AFX_NO_DB_SUPPORT
#define _AFX_NO_DAO_SUPPORT

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions

#ifndef _AFX_NO_OLE_SUPPORT
#include <afxole.h>         // MFC OLE classes
#include <afxdisp.h>        // MFC OLE automation classes
#endif // _AFX_NO_OLE_SUPPORT

#include <dbdao.h>
```

## README.TXT

```
ActiveX component that schedules a Broadcast Architecture show reminder.

This Readme contains the following sections:
* Summary Briefly describes the sample
* More InformationDetails how to compile and run the sample
* About SCHSAMPDescribes the sample in more detail.
* Using SCHSAMPExplains how to use SCHSAMP from a Web page.

For more information about how to write applications that schedule show reminders,

SUMMARY
=========
The SCHSAMP sample, Schsamp.dll, is an ActiveX component that schedules a show remi

SCHSAMP was developed using Visual Basic 5.0.
```

```
MORE INFORMATION
================
Before you can compile or use this sample, you must first install the
Broadcasting Architecture. This installs and registers
Microsoft Television System Services and TV Viewer, both of which are
required for SCHSAMP to run.

The following information describes the SCHSAMP sample.

To Compile SCHSAMP
-------------------
Open the SCHSAMP project file, Schsamp.vdp, in Visual Basic 5.0.
From the File menu, select Make SchSamp.dll.

To compile the component into a .Cab file for distribution
on the internet, use the Application Setup Wizard provided with
Visual Basic 5.0. Choose Create Internet Download Setup.

This control is unsigned. If you wish to distribute a version of
SCHSAMP on the Web you should sign the component. For instructions on
how to sign a .Cab file, see the Internet Client SDK.

To Set Up the Programming Environment
----------------------------------------------------
First, create a new ActiveX DLL project. Set the class to global, multiuse. Then se
You must also set a reference to Microsoft DAO 3.5 Object library. This enables you

To Run SCHSAMP
---------------
The SCHSAMP sample includes a version of the component that has already
been compiled into a .Cab file for delivery through a web page. The .Cab
file has not been signed, and thus in order to run the sample you must set
security in Internet Explorer 4.0 to allow unsigned controls to run.

To do this, go to the View menu and select Internet Options. Click the
Security tab.  Set Security settings for the appropriate zone.  If you plan
to run SCHSAMP from your machine, this should be Local Intranet zone.

Once the security is properly set, open SHCSAMP.HTML with Internet Explorer 4.0.
The SCHSAMP component will automatically download and install on your machine.
To run the script that sets the show reminder, click the Schedule Reminder button.

-------NOTE --------NOTE ----------NOTE ----------NOTE ---------NOTE ---
Note that you cannot set a reminder for a show that has already occurred.
The HTML Web currently is scripted to set a reminder for the show "Maid to Order"
on 8/5/97 at 3:00PM. If the current date is after 8/5/97, the sample will not run.
To correct this, edit the following lines of the HTML file:

SchSamp.Episode = "Maid to Order"
SchSamp.Network = ""
SchSamp.Duration = "120"
SchSamp.ShowTime = "8/5/97 3:00:00 PM"

Change the preceeding lines to reflect a future episode.

For best results, choose an episode from the
TV Viewer program guide. This will enable SCHSAMP to match the episode in
the Guide database and obtain description and tuning information.
-------NOTE --------NOTE ----------NOTE ----------NOTE ---------NOTE ---

Because SCHSAMP uses the IScheduledItems interface, you cannot use the
Search page of TV Viewer to view a sample set with SCHSAMP. To test whether
the reminder was set, follow the instructions detailed in Schsamp.htm.
```

```
SCHSAMP Files
--------------
SCHSAMP.HTML is an HTML/VBScript file uses SCHSAMP to set a show reminder.

SCHSAMP.CAB contains a compiled version of SCHSAMP.

SCHSAMP.VBP is the Visual Basic project file.

SCHSAMP.CLS contains the Visual Basic class module

ABOUT SCHSAMP
===============
SchSamp is an ActiveX component that schedules a TV Viewer reminder for a broadcast
This component could be used, for example, on an television enhancement page to ena

-------NOTE --------NOTE ----------NOTE ----------NOTE ---------NOTE ---
Reminders that are set by any means other than ITVViewer::SetReminder cannot be vie
-------NOTE --------NOTE ----------NOTE ----------NOTE ---------NOTE ---

USING SCHSAMP
==============
Because SchSamp is implemented as an ActiveX component, it can be called from a var

-------NOTE --------NOTE ----------NOTE ----------NOTE ---------NOTE ---
The SchSamp component is not signed. In order to run the sample you must set securi
-------NOTE --------NOTE ----------NOTE ----------NOTE ---------NOTE ---

The following topics describe how to create an instance of SchSamp, pass in episode

Inserting SCHSAMP in a Web Page
------------------------------------------------------
You can call the SchSamp component from a Web page. To create an instance of the cc

<OBJECT ID="MySchSamp"
CLASSID="CLSID:BE521C45-08DA-11D1-98AE-080009DC95C5"
CODEBASE="SchSamp.CAB#version=1,0,0,0">
</OBJECT>


Set the ID parameter to specify a name for this instance of the control.  In the pr

Specifying Episode Data
------------------------------
The SchSamp component contains the following properties.

PropertyData typeDescription
----------------------------------------
EpisodeStringTitle of the episode.
NetworkStringIf applicable, the network on which the episode appears.
DurationStringLength of the episode, in minutes.
ShowTimeStringTime and date that the episode starts.
PreTimeLongThe number of minutes early that the reminder should run.

These properties store data about the episode for which you wish to set a reminder.
The following example uses VBScript in a Web page to set values for the SchSamp prc

<SCRIPT LANGUAGE = VBScript>
Sub SetIt_OnClick
      'set the show's properties
      SchSamp.Episode = "Maid to Order"
      SchSamp.Network = ""
      SchSamp.Duration = "120"
      SchSamp.ShowTime = "8/5/97 3:00:00 PM"
      SchSamp.PreTime = 5

      'Schedule the reminder
      SchSamp.SetReminder
End Sub
</SCRIPT>

Note that you cannot set a reminder for a show that has already occurred. The HTML
For best results, choose an episode from the TV Viewer program guide. This will ena

Scheduling the Show Reminder
---------------------------------------
Once the properties of SchSamp are set as described in the Broadcast Architecture S

<SCRIPT LANGUAGE = VBScript>
  'Schedule the reminder
   SchSamp.SetReminder
</SCRIPT>
```

# README.TXT

```
An MFC sample that connects to and controls TV Viewer.

Summary
=======
TVXSamp is an MFC application that connects to and controls an instance of TV Viewe

For more information, see `` TV Viewer `` and `` Creating TV Viewer Controls ``
in the Broadcast Architecture Programmer's Reference.

More Information
================
The following information describes the TVXSamp sample.


To Compile TVXSamp
------------------
From the command prompt, use the following command:

nmake


To Run TVXSamp
--------------
You must start TV Viewer before you run TVXSamp.exe. Otherwise, TVXSamp will not be

Start TVXSamp.exe either from the command line by typing "TVXSamp", or by double-cl

[Toggle TV Mode]
Toggles TV Viewer between full screen and desktop mode.

[Tune to TV Config]
Tunes TV Viewer to the TV Config channel, 1.
```

```
[Tune Back to Previous Channel]
Tunes TV Viewer to the previous channel. This is the same
functionality as a Back button in a Web browser.


TVXSamp Files
=============

Tvdisp.h
This is the header file for the TV Viewer dispatch interface.
It is created from Tvdisp.odl

TVXSamp.h
    This is the main header file for the application.  It includes other
    project specific headers (including Resource.h) and declares the
    CTVXSampApp application class.

TVXSamp.cpp
    This is the main application source file that contains the application
    class CTVXSampApp.

TVXSamp.rc
    This is a listing of all of the Microsoft Windows resources that the
    program uses.  It includes the icons, bitmaps, and cursors that are stored
    in the RES subdirectory.  This file can be directly edited in Microsoft
Developer Studio.

res\TVXSamp.ico
    This is an icon file, which is used as the application's icon.  This
    icon is included by the main resource file TVXSamp.rc.

res\TVXSamp.rc2
    This file contains resources that are not edited by Microsoft
Developer Studio.  You should place all resources not
editable by the resource editor in this file.

TVXSamp.reg
    This is an example .REG file that shows you the kind of registration
    settings the framework will set for you.  You can use this as a .REG
    file to go along with your application.

TVXSamp.odl
    This file contains the Object Description Language source code for the
    type library of your application.

TVXSamp.clw
    This file contains information used by ClassWizard to edit existing
    classes or add new classes.  ClassWizard also uses this file to store
    information needed to create and edit message maps and dialog data
    maps and to create prototype member functions.


/////////////////////////////////////////////////////////////////////////////

AppWizard creates one dialog class and automation proxy class:

TVXSampDlg.h, TVXSampDlg.cpp - the dialog
    These files contain your CTVXSampDlg class.  This class defines
    the behavior of your application's main dialog.  The dialog's
    template is in TVXSamp.rc, which can be edited in Microsoft
Developer Studio.
```

```
DlgProxy.h, DlgProxy.cpp - the automation object
    These files contain your CTVXSampDlgAutoProxy class.  This class
    is called the "automation proxy" class for your dialog, because it
    takes care of exposing the automation methods and properties that
    automation controllers can use to access your dialog.  These methods
    and properties are not exposed from the dialog class directly, because
    in the case of a modal dialog-based MFC application it is cleaner and
    easier to keep the OLE automation object separate from the user interface.


/////////////////////////////////////////////////////////////////////////////
Other standard files:

StdAfx.h, StdAfx.cpp
    These files are used to build a precompiled header (PCH) file
    named TVXSamp.pch and a precompiled types file named StdAfx.obj.

Resource.h
    This is the standard header file, which defines new resource IDs.
    Microsoft Developer Studio reads and updates this file.
```

# DLGPROXY.H

```
// DlgProxy.h : TV Viewer sample application
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
// implementation file
//
//

#if !defined(AFX_DLGPROXY_H__FF52102B_0CE4_11D1_98AE_080009DC95C5__INCLUDED_)
#define AFX_DLGPROXY_H__FF52102B_0CE4_11D1_98AE_080009DC95C5__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class CTVXSampDlg;

/////////////////////////////////////////////////////////////////////////////
// CTVXSampDlgAutoProxy command target

class CTVXSampDlgAutoProxy : public CCmdTarget
{
DECLARE_DYNCREATE(CTVXSampDlgAutoProxy)

CTVXSampDlgAutoProxy();           // protected constructor used by dynamic creation

// Attributes
public:
```

```
CTVXSampDlg* m_pDialog;

// Operations
public:

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CTVXSampDlgAutoProxy)
public:
virtual void OnFinalRelease();
//}}AFX_VIRTUAL

// Implementation
protected:
virtual ~CTVXSampDlgAutoProxy();

// Generated message map functions
//{{AFX_MSG(CTVXSampDlgAutoProxy)
// NOTE - the ClassWizard will add and remove member functions here.
//}}AFX_MSG

DECLARE_MESSAGE_MAP()
DECLARE_OLECREATE(CTVXSampDlgAutoProxy)

// Generated OLE dispatch map functions
//{{AFX_DISPATCH(CTVXSampDlgAutoProxy)
// NOTE - the ClassWizard will add and remove member functions here.
//}}AFX_DISPATCH
DECLARE_DISPATCH_MAP()
DECLARE_INTERFACE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately befor

#endif // !defined(AFX_DLGPROXY_H__FF52102B_0CE4_11D1_98AE_080009DC95C5__INCLUDED_)
```

## DLGPROXY.CPP

```
// DlgProxy.cpp : TV Viewer sample application
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
// implementation file
//

#include "stdafx.h"
#include "TVXSamp.h"
#include "DlgProxy.h"
#include "TVXSampDlg.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CTVXSampDlgAutoProxy

IMPLEMENT_DYNCREATE(CTVXSampDlgAutoProxy, CCmdTarget)

CTVXSampDlgAutoProxy::CTVXSampDlgAutoProxy()
{
EnableAutomation();

// To keep the application running as long as an OLE automation
//object is active, the constructor calls AfxOleLockApp.
AfxOleLockApp();

// Get access to the dialog through the application's
//  main window pointer.  Set the proxy's internal pointer
//  to point to the dialog, and set the dialog's back pointer to
//  this proxy.
ASSERT (AfxGetApp()->m_pMainWnd != NULL);
ASSERT_VALID (AfxGetApp()->m_pMainWnd);
ASSERT_KINDOF(CTVXSampDlg, AfxGetApp()->m_pMainWnd);
m_pDialog = (CTVXSampDlg*) AfxGetApp()->m_pMainWnd;
m_pDialog->m_pAutoProxy = this;
}

CTVXSampDlgAutoProxy::~CTVXSampDlgAutoProxy()
{
// To terminate the application when all objects created with
// with OLE automation, the destructor calls AfxOleUnlockApp.
//  Among other things, this will destroy the main dialog
AfxOleUnlockApp();
}

void CTVXSampDlgAutoProxy::OnFinalRelease()
{
// When the last reference for an automation object is released
// OnFinalRelease is called.  The base class will automatically
// deletes the object.  Add additional cleanup required for your
// object before calling the base class.

CCmdTarget::OnFinalRelease();
}

BEGIN_MESSAGE_MAP(CTVXSampDlgAutoProxy, CCmdTarget)
//{{AFX_MSG_MAP(CTVXSampDlgAutoProxy)
// NOTE - the ClassWizard will add and remove mapping macros here.
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

BEGIN_DISPATCH_MAP(CTVXSampDlgAutoProxy, CCmdTarget)
//{{AFX_DISPATCH_MAP(CTVXSampDlgAutoProxy)
// NOTE - the ClassWizard will add and remove mapping macros here.
//}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()

// Note: we add support for IID_ITVXSamp to support typesafe binding
//  from VBA.  This IID must match the GUID that is attached to the
//  dispinterface in the .ODL file.

// {FF521024-0CE4-11D1-98AE-080009DC95C5}
static const IID IID_ITVXSamp =
{ 0xff521024, 0xce4, 0x11d1, { 0x98, 0xae, 0x8, 0x0, 0x9, 0xdc, 0x95, 0xc5 } };

BEGIN_INTERFACE_MAP(CTVXSampDlgAutoProxy, CCmdTarget)
INTERFACE_PART(CTVXSampDlgAutoProxy, IID_ITVXSamp, Dispatch)
END_INTERFACE_MAP()

// The IMPLEMENT_OLECREATE2 macro is defined in StdAfx.h of this project
// {FF521022-0CE4-11D1-98AE-080009DC95C5}
IMPLEMENT_OLECREATE2(CTVXSampDlgAutoProxy, "TVXSamp.Application", 0xff521022, 0xce4

/////////////////////////////////////////////////////////////////////////////
// CTVXSampDlgAutoProxy message handlers
```

## RESOURCE.H

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by tvxsamp.rc
//
#define IDP_OLE_INIT_FAILED             100
#define IDD_TVXSAMP_DIALOG              102
#define IDR_MAINFRAME                   128
#define IDC_BUTTON1                     1000
#define IDC_BUTTON2                     1001
#define IDC_BUTTON4                     1003

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        129
#define _APS_NEXT_COMMAND_VALUE         32771
#define _APS_NEXT_CONTROL_VALUE         1004
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

## STDAFX.CPP

```
// stdafx.cpp : TV Viewer sample application
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
// implementation file
//
//
//TVXSamp.pch will be the pre-compiled header
//stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"
```

## STDAFX.H

```
// stdafx.h : TV Viewer sample application
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//

#if !defined(AFX_STDAFX_H__FF52102E_0CE4_11D1_98AE_080009DC95C5__INCLUDED_)
#define AFX_STDAFX_H__FF52102E_0CE4_11D1_98AE_080009DC95C5__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#define VC_EXTRALEAN// Exclude rarely-used stuff from Windows headers

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#include <afxdisp.h>        // MFC OLE automation classes
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>// MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT


// This macro is the same as IMPLEMENT_OLECREATE, except it passes TRUE
//  for the bMultiInstance parameter to the COleObjectFactory constructor.
//  We want a separate instance of this application to be launched for
//  each OLE automation proxy object requested by automation controllers.
#ifndef IMPLEMENT_OLECREATE2
#define IMPLEMENT_OLECREATE2(class_name, external_name, l, w1, w2, b1, b2, b3, b4,
AFX_DATADEF COleObjectFactory class_name::factory(class_name::guid, \
RUNTIME_CLASS(class_name), TRUE, _T(external_name)); \
const AFX_DATADEF GUID class_name::guid = \
{ l, w1, w2, { b1, b2, b3, b4, b5, b6, b7, b8 } };
#endif // IMPLEMENT_OLECREATE2

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately befor

#endif // !defined(AFX_STDAFX_H__FF52102E_0CE4_11D1_98AE_080009DC95C5__INCLUDED_)
```

## TVDISP.CPP

```
// tvdisp.cpp : TV Viewer sample application
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//

#include "stdafx.h"
#include "tvdisp.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif



/////////////////////////////////////////////////////////////////////////////
// ITVViewer properties

/////////////////////////////////////////////////////////////////////////////
// ITVViewer operations

void ITVViewer::SetTVMode(BOOL fTVMode)
{
static BYTE parms[] =
VTS_BOOL;
InvokeHelper(0xfa1, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 fTVMode);
}

BOOL ITVViewer::IsTVMode()
{
BOOL result;
InvokeHelper(0xfa2, DISPATCH_METHOD, VT_BOOL, (void*)&result, NULL);
return result;
}

BOOL ITVViewer::IsChannelBarUp()
{
BOOL result;
InvokeHelper(0xfa3, DISPATCH_METHOD, VT_BOOL, (void*)&result, NULL);
return result;
}

BOOL ITVViewer::IsModalDialogUp()
{
BOOL result;
InvokeHelper(0xfa4, DISPATCH_METHOD, VT_BOOL, (void*)&result, NULL);
return result;
}
```

```
BOOL ITVViewer::IsLoaderActive()
{
BOOL result;
InvokeHelper(0xfa5, DISPATCH_METHOD, VT_BOOL, (void*)&result, NULL);
return result;
}

DATE ITVViewer::GlobalStartTime()
{
DATE result;
InvokeHelper(0xfa6, DISPATCH_METHOD, VT_DATE, (void*)&result, NULL);
return result;
}

DATE ITVViewer::GlobalEndTime()
{
DATE result;
InvokeHelper(0xfa7, DISPATCH_METHOD, VT_DATE, (void*)&result, NULL);
return result;
}

LPUNKNOWN ITVViewer::ChannelList()
{
LPUNKNOWN result;
InvokeHelper(0xfa8, DISPATCH_METHOD, VT_UNKNOWN, (void*)&result, NULL);
return result;
}

long ITVViewer::ViewerID()
{
long result;
InvokeHelper(0xfa9, DISPATCH_METHOD, VT_I4, (void*)&result, NULL);
return result;
}

void ITVViewer::WantNumKeys(BOOL fWantNumKeys)
{
static BYTE parms[] =
VTS_BOOL;
InvokeHelper(0xfaa, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 fWantNumKeys);
}

void ITVViewer::Tune(long lTuningSpace, long lChannelNumber, long lVideoStream, lon
{
static BYTE parms[] =
VTS_I4 VTS_I4 VTS_I4 VTS_I4 VTS_BSTR;
InvokeHelper(0xfab, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 lTuningSpace, lChannelNumber, lVideoStream, lAudioStream, bsIPStream);
}

void ITVViewer::GetCurrentTuningInfo(long* lTuningSpace, long* lChannelNumber, long
{
static BYTE parms[] =
VTS_PI4 VTS_PI4 VTS_PI4 VTS_PI4 VTS_PBSTR;
InvokeHelper(0xfac, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 lTuningSpace, lChannelNumber, lVideoStream, lAudioStream, pbsIPAddress);
}

void ITVViewer::GetPreviousTuningInfo(long* lTuningSpace, long* lChannelNumber, lon
{
static BYTE parms[] =
VTS_PI4 VTS_PI4 VTS_PI4 VTS_PI4 VTS_PBSTR;
```

```
InvokeHelper(0xfad, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 lTuningSpace, lChannelNumber, lVideoStream, lAudioStream, pbsIPAddress);
}

void ITVViewer::SetReminder(LPUNKNOWN pEpisode, BOOL bRecord)
{
static BYTE parms[] =
VTS_UNKNOWN VTS_BOOL;
InvokeHelper(0xfae, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 pEpisode, bRecord);
}

BOOL ITVViewer::HasReminder(LPUNKNOWN pEpisode, BOOL bRecord)
{
BOOL result;
static BYTE parms[] =
VTS_UNKNOWN VTS_BOOL;
InvokeHelper(0xfaf, DISPATCH_METHOD, VT_BOOL, (void*)&result, parms,
pEpisode, bRecord);
return result;
}

void ITVViewer::DeleteReminder(LPUNKNOWN pEpisode, BOOL bRecord)
{
static BYTE parms[] =
VTS_UNKNOWN VTS_BOOL;
InvokeHelper(0xfb0, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 pEpisode, bRecord);
}

BOOL ITVViewer::HasEnhancement(LPUNKNOWN pEpisode)
{
BOOL result;
static BYTE parms[] =
VTS_UNKNOWN;
InvokeHelper(0xfb1, DISPATCH_METHOD, VT_BOOL, (void*)&result, parms,
pEpisode);
return result;
}

BOOL ITVViewer::IsCC()
{
BOOL result;
InvokeHelper(0xfb2, DISPATCH_METHOD, VT_BOOL, (void*)&result, NULL);
return result;
}


/////////////////////////////////////////////////////////////////////////////
// ITVControl properties

/////////////////////////////////////////////////////////////////////////////
// ITVControl operations

BOOL ITVControl::OnIdle()
{
BOOL result;
InvokeHelper(0xbb9, DISPATCH_METHOD, VT_BOOL, (void*)&result, NULL);
return result;
}

void ITVControl::Tune(long ltsNew, long lcnNew, long lvsNew, long lasNew, LPCTSTR b
{
```

```
static BYTE parms[] =
VTS_I4 VTS_I4 VTS_I4 VTS_I4 VTS_BSTR VTS_I4 VTS_I4 VTS_I4 VTS_I4 VTS_BSTR;
InvokeHelper(0xbba, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 ltsNew, lcnNew, lvsNew, lasNew, bsIPNew, ltsPrev, lcnPrev, lvsPrev, lasPrev, bsIPP
}

void ITVControl::TearDown()
{
InvokeHelper(0xbbb, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

void ITVControl::SyncEvent(long iEvent, LPCTSTR pParm1, LPCTSTR pParm2)
{
static BYTE parms[] =
VTS_I4 VTS_BSTR VTS_BSTR;
InvokeHelper(0xbbc, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 iEvent, pParm1, pParm2);
}

void ITVControl::EpisodeStatusChanged(long iChange, LPUNKNOWN pEpi)
{
static BYTE parms[] =
VTS_I4 VTS_UNKNOWN;
InvokeHelper(0xbbd, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 iChange, pEpi);
}

void ITVControl::PowerChange(BOOL bPowerOn, BOOL bUIAllowed)
{
static BYTE parms[] =
VTS_BOOL VTS_BOOL;
InvokeHelper(0xbbf, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 bPowerOn, bUIAllowed);
}

void ITVControl::OnTVFocus()
{
InvokeHelper(0xbc0, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

void ITVControl::SetOutput(LPCTSTR bsDeviceName)
{
static BYTE parms[] =
VTS_BSTR;
InvokeHelper(0xbc1, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 bsDeviceName);
}

BOOL ITVControl::GetCC()
{
BOOL result;
InvokeHelper(0xbc2, DISPATCH_METHOD, VT_BOOL, (void*)&result, NULL);
return result;
}

void ITVControl::SetCC(BOOL bCC)
{
static BYTE parms[] =
VTS_BOOL;
InvokeHelper(0xbc3, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 bCC);
}
```

## TVDISP.H

```
// tvdisp.h : TV Viewer sample application
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//
/////////////////////////////////////////////////////////////////////////////
// ITVViewer wrapper class

class ITVViewer : public COleDispatchDriver
{
public:
ITVViewer() {}// Calls COleDispatchDriver default constructor
ITVViewer(LPDISPATCH pDispatch) : COleDispatchDriver(pDispatch) {}
ITVViewer(const ITVViewer& dispatchSrc) : COleDispatchDriver(dispatchSrc) {}

// Attributes
public:

// Operations
public:
void SetTVMode(BOOL fTVMode);
BOOL IsTVMode();
BOOL IsChannelBarUp();
BOOL IsModalDialogUp();
BOOL IsLoaderActive();
DATE GlobalStartTime();
DATE GlobalEndTime();
LPUNKNOWN ChannelList();
long ViewerID();
void WantNumKeys(BOOL fWantNumKeys);
void Tune(long lTuningSpace, long lChannelNumber, long lVideoStream, long lAudioStr
void GetCurrentTuningInfo(long* lTuningSpace, long* lChannelNumber, long* lVideoStr
void GetPreviousTuningInfo(long* lTuningSpace, long* lChannelNumber, long* lVideoSt
void SetReminder(LPUNKNOWN pEpisode, BOOL bRecord);
BOOL HasReminder(LPUNKNOWN pEpisode, BOOL bRecord);
void DeleteReminder(LPUNKNOWN pEpisode, BOOL bRecord);
BOOL HasEnhancement(LPUNKNOWN pEpisode);
BOOL IsCC();
};
/////////////////////////////////////////////////////////////////////////////
// ITVControl wrapper class

class ITVControl : public COleDispatchDriver
{
public:
ITVControl() {}// Calls COleDispatchDriver default constructor
ITVControl(LPDISPATCH pDispatch) : COleDispatchDriver(pDispatch) {}
ITVControl(const ITVControl& dispatchSrc) : COleDispatchDriver(dispatchSrc) {}
```

```
// Attributes
public:

// Operations
public:
BOOL OnIdle();
void Tune(long ltsNew, long lcnNew, long lvsNew, long lasNew, LPCTSTR bsIPNew, long
void TearDown();
void SyncEvent(long iEvent, LPCTSTR pParm1, LPCTSTR pParm2);
void EpisodeStatusChanged(long iChange, LPUNKNOWN pEpi);
void PowerChange(BOOL bPowerOn, BOOL bUIAllowed);
void OnTVFocus();
void SetOutput(LPCTSTR bsDeviceName);
BOOL GetCC();
void SetCC(BOOL bCC);
};
```

## TVDISPID.H

```
// tvdispid.h : TV Viewer sample application
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.


#ifndef __TVDISPID_H__
#define __TVDISPID_H__

#ifdef __MKTYPLIB__

// Sync event notifications
typedef enum tvsyncevent
{
//Loader sync events
//Corresponding messages from epgldrx.h
//EPGLDR_STARTING
//EPGLDR_ACTIVE_COMMIT_STARTING
//EPGLDR_ACTIVE_COMMIT_ENDING
//EPGLDR_PASSIVE_COMMIT_STARTING
//EPGLDR_PASSIVE_COMMIT_ENDING
//EPGLDR_ENDING

//Viewer sync events
keViewerLogin= 107,
keViewerChange= 108,
keCurrentViewerChannelListChange= 109,

//Other sync events
keSysTimeChange= 110
} TVSYNCEVENT;

// EpisodeStatusChanged notification enums
```

```
typedef enum episodestatus
{
keReminderStatus= 1,
kePurchaseStatus= 2,
keDSSEmailStatus= 3
} EPISODESTATUS;

#endif

// Dispatch IDs

// ITVControl

#define dispidOnIdle3001
#define dispidTuneControl3002
#define dispidTearDown3003
#define dispidSyncEvent3004
#define dispidEpisodeStatusChanged3005
#define dispidPowerChange3007
#define dispidOnTVFocus   3008
#define dispidTVSetOutput3009
#define dispidGetCC   3010
#define dispidSetCC   3011

// ITVExplorer

#define dispidSetTVMode4001
#define dispidIsTVMode4002
#define dispidIsChannelBarUp4003
#define dispidIsModalDialogUp4004
#define dispidIsLoaderActive4005
#define dispidGlobalStartTime4006
#define dispidGlobalEndTime4007
#define dispidChannelList4008
#define dispidViewerID4009
#define dispidWantNumKeys4010
#define dispidTVXTune4011
#define dispidGetCurrentTuningInfo4012
#define dispidGetPreviousTuningInfo4013
#define dispidSetReminder4014
#define dispidHasReminder4015
#define dispidDeleteReminder4016
#define dispidHasEnhancement            4017
#define dispidIsCC                      4018

#endif // __TVDISPID_H__
```

## TVXSAMP.CPP

```
//---------------------------------------------------------------------------
// TVXSampDlg.cpp : TV Viewer sample application
//---------------------------------------------------------------------------
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For more information about writing applications that interact
// with TV Viewer, see `` Creating TV Viewer Controls ``
// in the Broadcast Architecture Programmer's Reference.
//
//

#include "stdafx.h"
#include "TVXSamp.h"
#include "TVXSampDlg.h"
#include "Tvdisp.h"
#include <initguid.h>

DEFINE_GUID(IID_ITVViewer,0x3F8A2EA1L,0xC171,0x11CF,0x86,0x8C,0x00,0x80,0x5F,0x2C,0
DEFINE_GUID(IID_ITVDisp, 0x3F8A2EA1L, 0xC171,0x11cf,0x86,0x8C,0x00,0x80,0x5F,0x2C,0
DEFINE_GUID(CLSID_TVViewer,0x5543DD10L,0xB41D,0x11CF,0x86,0x82,0x00,0x80,0x5F,0x2C,

extern ITVViewer *TVX;

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CTVXSampApp

BEGIN_MESSAGE_MAP(CTVXSampApp, CWinApp)
//{{AFX_MSG_MAP(CTVXSampApp)
// NOTE - the ClassWizard will add and remove mapping macros here.
//    DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG
ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CTVXSampApp construction

CTVXSampApp::CTVXSampApp()
{
// TODO: add construction code here,
// Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CTVXSampApp object

CTVXSampApp theApp;

/////////////////////////////////////////////////////////////////////////////
// CTVXSampApp initialization

BOOL CTVXSampApp::InitInstance()
{
// Initialize OLE libraries
if (!AfxOleInit())
{
AfxMessageBox(IDP_OLE_INIT_FAILED);
return FALSE;
}

AfxEnableControlContainer();
```

```
//--------------< code that gets a reference to TV Viewer >-------------
//--------------------------------------------------------------------------------
/*
The following code gets a reference to TV Viewer which
TVXSamp uses to call the ITVViewer methods.
This is implemented during TVXSamp's initialization
to ensure that it connects to TV Viewer before
calling the ITVViewer methods.

TV Viewer must be running or else the following code will fail.
*/
HRESULT hr = 0;
IUnknown *punk;
IDispatch *dispatch;

hr = GetActiveObject(CLSID_TVViewer,NULL,&punk);
if (SUCCEEDED(hr))
{
hr=punk->QueryInterface(IID_IDispatch,(void**) &dispatch);
punk->Release();
if (SUCCEEDED(hr))
{
TVX=new ITVViewer(dispatch);
}
}


// Standard initialization
// If you are not using these features and wish to reduce the size
//  of your final executable, you should remove from the following
//  the specific initialization routines you do not need.

// Parse the command line to see if launched as OLE server
if (RunEmbedded() || RunAutomated())
{
// Register all OLE server (factories) as running.  This enables the
//  OLE libraries to create objects from other applications.
COleTemplateServer::RegisterAll();
}
else
{
// When a server application is launched stand-alone, it is a good idea
//  to update the system registry in case it has been damaged.
COleObjectFactory::UpdateRegistryAll();
}

CTVXSampDlg dlg;
m_pMainWnd = &dlg;
int nResponse = dlg.DoModal();
if (nResponse == IDOK)
{
// TODO: Place code here to handle when the dialog is
//  dismissed with OK


}
else if (nResponse == IDCANCEL)
{
// TODO: Place code here to handle when the dialog is
//  dismissed with Cancel

}
```

```
// Since the dialog has been closed, return FALSE so that we exit the
//  application, rather than start the application's message pump.
return FALSE;
}
```

## TVXSAMP.H

```
// TVXSamp.h : TV Viewer sample application
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//
//
//

#if !defined(AFX_TVXSAMP_H__FF521027_0CE4_11D1_98AE_080009DC95C5__INCLUDED_)
#define AFX_TVXSAMP_H__FF521027_0CE4_11D1_98AE_080009DC95C5__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"// main symbols

/////////////////////////////////////////////////////////////////////////////
// CTVXSampApp:
// See TVXSamp.cpp for the implementation of this class
//

class CTVXSampApp : public CWinApp
{
public:
CTVXSampApp();

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CTVXSampApp)
public:
virtual BOOL InitInstance();
//}}AFX_VIRTUAL

// Implementation

//{{AFX_MSG(CTVXSampApp)
// NOTE - the ClassWizard will add and remove member functions here.
//    DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};


/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately befor

#endif // !defined(AFX_TVXSAMP_H__FF521027_0CE4_11D1_98AE_080009DC95C5__INCLUDED_)
```

## TVXSAMPDLG.CPP

```
//-------------------------------------------------------------------------------
// TVXSampDlg.cpp : TV Viewer sample application
//-------------------------------------------------------------------------------
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For more information about writing applications that interact
// with TV Viewer, see `` Creating TV Viewer Controls ``
// in the Broadcast Architecture Programmer's Reference.
//
//

#include "stdafx.h"
#include "TVXSamp.h"
#include "TVXSampDlg.h"
#include "DlgProxy.h"
#include "Tvdisp.h"
#include <atlbase.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

ITVViewer *TVX;

/////////////////////////////////////////////////////////////////////////////
// CTVXSampDlg dialog

IMPLEMENT_DYNAMIC(CTVXSampDlg, CDialog);

CTVXSampDlg::CTVXSampDlg(CWnd* pParent /*=NULL*/)
: CDialog(CTVXSampDlg::IDD, pParent)
{
//{{AFX_DATA_INIT(CTVXSampDlg)
// NOTE: the ClassWizard will add member initialization here
//}}AFX_DATA_INIT
// Note that LoadIcon does not require a subsequent DestroyIcon in Win32
m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
m_pAutoProxy = NULL;
}
```

```
CTVXSampDlg::~CTVXSampDlg()
{
// If there is an automation proxy for this dialog, set
//  its back pointer to this dialog to NULL, so it knows
//  the dialog has been deleted.
if (m_pAutoProxy != NULL)
m_pAutoProxy->m_pDialog = NULL;
}

void CTVXSampDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CTVXSampDlg)
// NOTE: the ClassWizard will add DDX and DDV calls here
//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CTVXSampDlg, CDialog)
//{{AFX_MSG_MAP(CTVXSampDlg)
ON_WM_PAINT()
ON_WM_QUERYDRAGICON()
ON_WM_CLOSE()
ON_BN_CLICKED(IDC_BUTTON1, OnButton1)
ON_BN_CLICKED(IDC_BUTTON2, OnButton2)
ON_BN_CLICKED(IDC_BUTTON4, OnButton4)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CTVXSampDlg message handlers

BOOL CTVXSampDlg::OnInitDialog()
{
CDialog::OnInitDialog();

// Set the icon for this dialog.  The framework does this automatically
//  when the application's main window is not a dialog
SetIcon(m_hIcon, TRUE);// Set big icon
SetIcon(m_hIcon, FALSE);// Set small icon

// TODO: Add extra initialization here

return TRUE;  // return TRUE  unless you set the focus to a control
}

// If you add a minimize button to your dialog, you will need the code below
//  to draw the icon.  For MFC applications using the document/view model,
//  this is automatically done for you by the framework.

void CTVXSampDlg::OnPaint()
{
if (IsIconic())
{
CPaintDC dc(this); // device context for painting

SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

// Center icon in client rectangle
int cxIcon = GetSystemMetrics(SM_CXICON);
int cyIcon = GetSystemMetrics(SM_CYICON);
CRect rect;
GetClientRect(&rect);
int x = (rect.Width() - cxIcon + 1) / 2;
int y = (rect.Height() - cyIcon + 1) / 2;

// Draw the icon
dc.DrawIcon(x, y, m_hIcon);
}
else
{
CDialog::OnPaint();
}
}

// The system calls this to obtain the cursor to display while the user drags
//  the minimized window.
HCURSOR CTVXSampDlg::OnQueryDragIcon()
{
return (HCURSOR) m_hIcon;
}

// Automation servers should not exit when a user closes the UI
//  if a controller still holds on to one of its objects.  These
//  message handlers make sure that if the proxy is still in use,
//  then the UI is hidden but the dialog remains around if it
//  is dismissed.

void CTVXSampDlg::OnClose()
{
if (CanExit())
CDialog::OnClose();
}

void CTVXSampDlg::OnOK()
{
if (CanExit())
CDialog::OnOK();
}

void CTVXSampDlg::OnCancel()
{
if (CanExit())
CDialog::OnCancel();
}

BOOL CTVXSampDlg::CanExit()
{
// If the proxy object is still around, then the automation
//  controller is still holding on to this application.  Leave
//  the dialog around, but hide its UI.
if (m_pAutoProxy != NULL)
{
ShowWindow(SW_HIDE);
return FALSE;
}

return TRUE;
}

//-----------< event handler that toggles TV Viewer mode >-----------
//---------------------------------------------------------------------------------
/*
The following code implements an event handler
that toggles TV Viewer between desktop and full
screen mode when the user clicks Button1. It checks
which mode TV Viewer is displaying in, and toggles
```

```
it to the other mode. For example, if TV Viewer is
in desktop mode, this method sets it to full screen
mode and vide versa.
*/

void CTVXSampDlg::OnButton1()
{

//check whether TV Viewer is in
//full screen mode
if (TVX->IsTVMode())
{
//if it is,
//change the mode to desktop
TVX->SetTVMode(false);
}
else
{
//if it is not,
//change the mode to full screen
TVX->SetTVMode(true);
}

}


//-----< event handler that tunes TV Viewer to a new channel >------
//----------------------------------------------------------------------------------
/*
The following code implements an event handler
that tunes TV Viewer to the TV configuration
channel when they click Button2.

The TV configuration channel was chosen for
this example because it is installed with TV Viewer
and is present on all client machines.
*/
void CTVXSampDlg::OnButton2()
{

//Tune TVViewer to channel 1 of tuning space -2
//the audio and video substreams have been set to
//-1, causing TV Viewer to use the default values.

TVX->Tune(-2,1,-1,-1,NULL);

}


//---------< event handler that tunes to a previous channel >-----------
//----------------------------------------------------------------------------------
/*
The following code implements an event handler that
tunes TV Viewer to the previously displayed channel
when the user clicks Button4.
*/

void CTVXSampDlg::OnButton4()
{
long lTuningSpacePrev;
long lChannelNumberPrev;
long lAudioStreamPrev;
long lVideoStreamPrev;
BSTR bstrIPAddressPrev;

//get the tuning information about the previous channel
//from TV Viewer
TVX->GetPreviousTuningInfo(&lTuningSpacePrev, &lChannelNumberPrev,
 &lVideoStreamPrev, &lAudioStreamPrev, &bstrIPAddressPrev);

if ((lTuningSpacePrev != NULL) && (lChannelNumberPrev != NULL))
{
//Tune TV Viewer to the previous channel
TVX->Tune(lTuningSpacePrev, lChannelNumberPrev,
lVideoStreamPrev, lAudioStreamPrev, (LPCTSTR) bstrIPAddressPrev);
}

}
```

## TVXSAMPDLG.H

```
// TVXSampDlg.h : TV Viewer sample application
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//
//
//


#if !defined(AFX_TVXSAMPDLG_H__FF521029_0CE4_11D1_98AE_080009DC95C5__INCLUDED_)
#define AFX_TVXSAMPDLG_H__FF521029_0CE4_11D1_98AE_080009DC95C5__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class CTVXSampDlgAutoProxy;

/////////////////////////////////////////////////////////////////////////////
// CTVXSampDlg dialog

class CTVXSampDlg : public CDialog
{
DECLARE_DYNAMIC(CTVXSampDlg);
friend class CTVXSampDlgAutoProxy;

// Construction
public:
CTVXSampDlg(CWnd* pParent = NULL);// standard constructor
virtual ~CTVXSampDlg();

// Dialog Data
//{{AFX_DATA(CTVXSampDlg)
enum { IDD = IDD_TVXSAMP_DIALOG };
// NOTE: the ClassWizard will add data members here
//}}AFX_DATA

// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CTVXSampDlg)
protected:
virtual void DoDataExchange(CDataExchange* pDX);// DDX/DDV support
//}}AFX_VIRTUAL

// Implementation
protected:
CTVXSampDlgAutoProxy* m_pAutoProxy;
HICON m_hIcon;

BOOL CanExit();

// Generated message map functions
//{{AFX_MSG(CTVXSampDlg)
virtual BOOL OnInitDialog();
afx_msg void OnPaint();
afx_msg HCURSOR OnQueryDragIcon();
afx_msg void OnClose();
virtual void OnOK();
virtual void OnCancel();
afx_msg void OnButton1();
afx_msg void OnButton2();
afx_msg void OnButton4();
afx_msg void OnSetReminder();
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately befor

#endif // !defined(AFX_TVXSAMPDLG_H__FF521029_0CE4_11D1_98AE_080009DC95C5__INCLUDED
```

# README.TXT

```
-------------------------------------------------------------------
    Microsoft Broadcast Architecture SDK UseVideo Readme File
                         August 1997
-------------------------------------------------------------------

             (c) Copyright Microsoft Corporation, 1997

CONTENTS
========

Using the Video ActiveX Control


SUMMARY
=======

The UseVideo samples demonstrate presenting audio and video
```

```
in an application, a Web browser, and in a control you create.
The following four programs are contained in the UseVideo sample:

1. Video Test Project (VidCntrl) demonstrates a project containing a
   standard Visual Basic form that hosts and manipulates an instance
   of the Video ActiveX control.

2. Video Web Page (WebTune) demonstrates embedding and using the
   Video ActiveX control in a Web page.

3. Video Test Control (Vid_Tune) demonstrates creating a control that
   contains the Video ActiveX control as a constituent control.

4. Video MFC Test describes building an application for Windows with
   C++ classes from the Microsoft Foundation Class Library (MFC).

In addition to presenting audio and video, these samples show how to embed
a Video ActiveX control in your program and how to select different devices
to provide input to such a control. If your computer contains a tuner card,
these samples show how to tune to a channel.

/////////////////////////////////////////////////////////////////////////////
MORE INFORMATION
================

The following information describes running the UseVideo samples.
For information on system requirements, see the Client Hardware Requirements
section of the the Broadcast Architecture Device-Driver Development Kit (DDK).

NOTE: For these samples to have maximum value, you should have the
      following components installed:

   1. Microsoft  Windows  98, including the optional TV Viewer component.

   2. Microsoft Visual Basic version 5.0. For more information on installing
      and using Visual Basic 5.0, see the Visual Basic 5.0 documentation.

   3. Microsoft Visual C++ version 5.0. For more information on installing
      and using Visual C++ 5.0, including the Microsoft Foundation Class
      (MFC) Library, see the Visual C++ 5.0 documentation.


/////////////////////////////////////////////////////////////////////////////
To Run VidCntrl
-----------------------------

1. Start Microsoft Visual Basic 5.0.

2. When the New Project dialog box appears, select the Existing tab and
   open the VidCntrl.vbp project file.

3. From the Run menu, select Start.

4. When the Video Control Test form appears in run mode, click the
   File Source (Async.) item in the list box to select it for the
   video control s input source. Enter the file name of a particular
   MPEG-1 file along with the required path to locate such file in
   the text box and click the Set Input command button to assign the
   file you specified as the input source for audio and video. Click the
   Play button to play the file, click the Pause button to temporarily
   suspend playing, click the Stop command button to stop the file.
   Click the Volume slider to change the intensity of the video control s
   sound and the Balance slider to change the video control s stereo balance.
```

```
5. If your computer contains a tuner card for analog television, click
   the WDM TvTuner item in the list box to select it for the video control s
   input source. Enter a channel number in the text box and click the
   Set Input button to tune your card to the channel number you specified.

VidCntrl Files
------------

VidCntrl.frm
    This file contains the form image in addition to the images of all the
    elements on the form and the code to manipulate the video control.
    The form's code uses the following properties and methods of the video control
    and other objects associated with video control:
    a. Declares and allocates storage for variables of BPCDevices and
       BPCDeviceBase object types.
    b. Obtains a reference to each device of the video control's Devices property.
    c. Obtains the name of each device with the BPCDeviceBase Name property.
    d. Determines if the computer contains any tuner cards that supports the tuning
       space for Analog Cable TV with the video control's TSDeviceCount method.
    e. If so, tunes to a channel with the video control's Tune method.
    f. Sets the video control's Input property equal to one of the devices in
       the video control's Devices property.
    g. Determines if the video control's input source is from a tuner with the
       BPCDeviceBase HasChannel property.
    h. If so, sets the channel number of the tuner with the BPCDeviceBase Channel
       property.
    i. Determines if the the video control's input source is from a file with the
       BPCDeviceBase HasFilename property.
    j. If so, sets the name of the file with the BPCDeviceBase filename property.
    k. For file sources, plays, pauses, stops, changes the volume and balance with
       video control's Run, Pause, Stop, Volume, and Balance properties respectivel

VidCntrl.vbp
    This file is the makefile for the project.


///////////////////////////////////////////////////////////////////////////////
To Run WebTune
-------------

1. Start Microsoft Internet Explorer 4.0 (IE4).

2. In IE4, select the menu command File, Open.
   When the Open dialog box appears, click on Browse and locate
   the Vid_Test.htm file, and then click Open. When the Open
   dialog box reappears, click OK.

3. When the Web page loads and IE4 displays the page's text and objects,
   enter the file name of a particular MPEG-1 file along with the required
   path to locate such file in the text box. Click the Set File button to
   assign the file you specified as the input source for audio and video
   and to play the file and click the Stop File button to stop the file.

4. If your computer contains a tuner card for analog television, enter
   a channel number in the text box and click the Set Chan button to
   tune your card to the channel number you specified.

WebTune File
------------

Vid_Test.htm
    This file is an HTML document that contains the text, input elements,
    the Video ActiveX control object, and code written in VBScript that
    manipulates the video control. The code enclosed in the SCRIPT tag
    uses all the same properties and methods as those used in the VidCntrl.frm
    code.


////////////////////////////////////////////////////////////////////////////
To Run Vid_Tune
-------------

1. Start Microsoft Visual Basic 5.0.

2. When the New Project dialog box appears, select the Existing tab and
   open the Vid_Tune.vbg group project file.

3. From the Run menu, select Start.

4. When the form for the video test control appears in run mode, enter
   the file name of a particular MPEG-1 file along with the required path
   to locate such file in the text box.

5. Click the File button to assign the MPEG-1 file you specified as the
   input source for audio and video and to play the file and click the Stop
   button to stop playing the file.

6. If your computer contains a tuner card for analog television, click any
   of the channel buttons to assign the WDM TvTuner as the input source for
   audio and video and to tune your card to the channel number specified by
   the button.

Vid_Tune Files
------------

Vid_Tune.ctl
    This file contains the images of all the elements on the control's
    designer and the code to manipulate the video control. The designer's code
    uses all the same properties and methods as those used in the VidCntrl.frm
    code. In addition, code in the designer exposes custom properties, creates
    and raises a click event, and creates the MyTune method.

Test_Vid.frm
    This file contains the form image, the image of the Video Test control,
    and the code to manipulate the Video Test control. The form's code contains
    the click event procedure that calls the Video Test control's MyTune method.

Vid_Tune.vbp
    This file is the makefile for the Video Test Control project.

Test_Vid.vbp
    This file is the makefile for the test project that verifies the Video
    Test Control.

Vid_Tune.vbg
    This file is the group file containing the Control project and the test
    project.


////////////////////////////////////////////////////////////////////////////
To Run VideoMFC
----------------------------

1. Start Microsoft Visual C++ 5.0.
```

```
2. From the File menu, select Open Workspace.

3. When the Open Workspace dialog box appears, go to the appropriate
   directory and open the VdMFC.mak file.

4. From the Build menu, select the Win32 Release configuration from
   the Set Active Configuration menu item. After setting the project
   configuration, select Build VdMFC.exe from the Build menu. After
   building the project, select Execute VdMFC.exe from the Build menu.

5. When the VdMFC application starts, click the File Source (Async.)
   item in the list box to select it for the Vid control's input source.
   Enter the file name of a particular MPEG-1 file along with the required
   path to locate such file in the edit box. Click the Set Input command
   button to assign the MPEG-1 file you specified as the input source for
   audio and video. Click the Set File Name and then Play command buttons
   to start playing the MPEG-1 file you specified. Click the Pause command
   button to temporarily suspend playing. Click the Stop command button to
   stop playing the file.

6. If your computer contains a tuner card for analog television, click the
   WDM TvTuner item in the list box and then the Set Input command button
   to select it for the video control's input source. Enter a channel number
   in the edit box and then click the Set Channels command button to tune
   your card to the channel number you specified. Click the On/Off Video
   command button to allow the Vid control to receive video or to prevent it
   from receiving video.

/////////////////////////////////////////////////////////////////////////////
To Run VideoMFC from the command line
-----------------------------

1. Open a Command Prompt window, switch to the directory on your
   computer that contains the makefile (VdMFC.mak) and the source files
   for the video MFC application, and then type the following exactly
   as shown including the case of the text:
   NMAKE /f "VdMFC.mak" CFG="VdMFC - Win32 Release"

2. To run the built VdMFC application, switch to the \Release directory
   created within the directory containing the source files and type VdMFC.

VideoMFC Files
------------
The MFC AppWizard created all but one of the skeleton files for the VdMFC
application. Most of the files did not require modification. The following
summaries of the contents of each of the files that make up VdMFC are divided
into either the group requiring modifications or not.

*********************
NOT CREATED BY MFC APPWIZARD
*********************
Vidtypes.h
    This file contains the definitions for all the constants and interfaces
    that make up the Vid control. This file was not generated by the MFC
    AppWizard. This file was added to the project.

*********************
NO EDITING REQUIRED
*********************
VdMFC.h
    Main header file for the application that includes other project-specific
    headers (including Resource.h) and defines the CVdMFCApp application class.
    This file was generated by the MFC AppWizard and did not require editing.
```

```
VdMFC.cpp
    This is the main application source file that contains the application
    class CVdMFCApp. This file was generated by the MFC AppWizard and did not
    require editing.

VdMFC.rc
    Lists all of the Microsoft Windows resources that the VdMFC application
    uses including the icon stored in the RES subdirectory. This file was
    generated by the MFC AppWizard and did not require editing.

res\VdMFC.ico
    This is an icon file, which is used as the application's icon.  This
    icon is included by the main resource file VdMFC.rc.

res\VdMFC.rc2
    This file contains resources that are not edited by Microsoft Developer
    Studio. This file was generated by the MFC AppWizard and did not
    require editing.

VdMFC.mak
    This file is the makefile for the project.

VdMFC.dsp
    This file is the project file.

StdAfx.h, StdAfx.cpp
    These files are used to build a precompiled header (PCH) file
    named VdMFC.pch and a precompiled types file named StdAfx.obj.
    These files were generated by the MFC AppWizard and did not
    require editing.

Resource.h
    This is the standard header file, which defines new resource IDs.
    This file was generated by the MFC AppWizard and did not
    require editing.

bpcvid.h, bpcvid.cpp
    These files contain the CBPCVid class, which defines the behavior
    of the object that provides streaming video functionality. These
    files were generated by the MFC AppWizard and did not require editing.

bpcbase.h, bpcbase.cpp
    These files contain the CBPCDeviceBase class, which defines the behavior
    of the object that represents either an input or an output device for
    the Vid control. These files were generated by the MFC AppWizard and
    did not require editing.

*********************
EDITING REQUIRED
*********************
VdMFCDlg.h, VdMFCDlg.cpp - the dialog
    These files contain the CVdMFCDlg class. This class defines the
    behavior of the application's main dialog. These files were generated
    by the MFC AppWizard. The following additions were made to these files:
    a. Includes the header files that define the CBPCVid, CBPCDevices, and
       CBPCDeviceBase classes and the data types and interfaces for the Vid
       control.
    b. ClassWizard was used to create variable names for the Vid control and
       for values typed in the edit box for setting channels (m_channel) and
       for strings typed in the edit box for setting file names (m_filename).
       The class definition was edited to create protected variables for the
       list box (box) and an available input or output device for the Vid
```

```
   control (m_id). The class definition was also edited to create a private
   variable for a pointer to the IBPCDeviceBase interface.
c. The constructor for this class initializes the channel and file name
   variable.
d. The DoDataExchange() method for this class associates the variable names
   with the appropriate elements.
e. The class definition declares methods to be associated with click events
   for elements of the user interface. The class implementation associates
   the click events with the ON_BN_CLICKED() macro.
f. The OnInitDialog() method for this class was edited to:
   Assign the list-box control of the user interface to the CListBox
   variable, box.
   Obtain pointers to unknown and enumerate-variant objects that are part
   of the MFC library.
   Declare and allocate a CBPCDevices object and assign the collection of
   all the available input and output devices for the Vid control to it.
   Assign the unknown object to the value returned from the custom
   CBPCDevices::GetNewEnum() method.
   Query the unknown object to obtain an enumerate-variant object and then
   release the unknown object.
   Enter a while loop set to repeat until there are no more variants in
   the enumerate-variant object.
   Get the next variant in the enumerate-variant object and obtain a dispatch
   for such a variant. Declare and allocate a CBPCDeviceBase object and assign
   the dispatch value of the obtained variant to such an object. Retrieve the
   name of the device-base object with the CBPCDeviceBase::GetName() method
   and add it to the list box. Clear the variant.
   After exiting the while loop, release the enumerate-variant object.
g. The OnSetInput() and OnSetOutput() methods for this class were implemented t
   Obtain pointers to unknown and enumerate-variant objects.
   Obtain an interface pointer to IBPCDeviceBase defined in Vidtypes.h.
   Declare and allocate a CBPCDevices object and assign the collection of
   all the available input and output devices for the Vid control to it.
   Assign the unknown object to the value returned from the custom
   CBPCDevices::GetNewEnum() method.
   Query the unknown object to obtain an enumerate-variant object and then
   release the unknown object.
   Enter a for loop set to repeat the number of times determined by the
   CBPCVid::GetDeviceCount() method.
   For each iteration of the loop:
     Obtain the next variant in the enumerate-variant object.
     Query the variant object to obtain an IBPCDeviceBase interface pointer.
     Get the name property of the device with the interface's get_Name() method
     Gets the selected item in the list box and then gets the string
     associated with the item.
     If this string is the same as the device name, assign the video control's
     input and output to that particular device with the SetInput() and
     SetOutput() methods of the CBPCVid class. Also, assign the device to
     the private member variable for the device, m_pDeviceBase.
h. The OnSetChannel() method for this class was implemented to dispatch a
   pointer to the selected item of the list box, then attach such pointer
   to a CBPCDeviceBase variable with the CBPCDeviceBase::AttachDispatch()
   method, and then assign the channel number specified in the edit box with
   the CBPCDeviceBase::SetChannel() method.
i. The OnVideoOn() method for this class was implemented to reverse the
   VideoOn value of the Vid control by first retrieving such value and then
   setting it to the opposite value.
j. The OnSetFile() method for this class was implemented to assign the file
   name specified in the edit box with the CBPCVid::SetFileName() method.
k. The OnPlay(), OnPause(), and OnStop() methods for this class were
   implemented to play, suspend play, and stop the file with the Run(),
   Pause(), and Stop() methods respectively of the CBPCVid class.
```

```
bpcdev.h, bpcdev.cpp
    These files contain the CBPCDevices class, which defines the behavior
    of the object that represents a collection of all the available input
    and output devices for the Vid control. These files were generated
    by the MFC AppWizard.
    A declaration and an implementation were added for the GetNewEnum()
    method. This method retrieves an unknown object that refers to an
    enumerate-variant object.
```

## VID_TUNE.TXT

```
The Vid_Tune directory contains a Visual Basic group comprising
a Visual Basic control and a Visual Basic project to test the
control. The Visual Basic control contains a Video ActiveX control
as a constituent control.
```

## VIDCNTRL.TXT

```
The VidCntrl directory contains a Visual Basic project
that uses the Video ActiveX control.
```

## VIDEOMFC.TXT

```
The VideoMFC directory contains a Visual C++ project
workspace that uses the Video ActiveX control.
```

## BPCBASE.H

```
//
// bpcbase.h: Defines the CBPCDeviceBase class
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//
#if !defined(AFX_BPCDEVICEBASE_H__4E088A12_0E75_11D1_A073_00A0C9054174__INCLUDED_)
```

```
#define AFX_BPCDEVICEBASE_H__4E088A12_0E75_11D1_A073_00A0C9054174__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
// Machine generated IDispatch wrapper class(es) created by Microsoft Visual C++

// NOTE: Do not modify the contents of this file.  If this class is regenerated by
//  Microsoft Visual C++, your modifications will be overwritten.

/////////////////////////////////////////////////////////////////////////////
// CBPCDeviceBase wrapper class

class CBPCDeviceBase : public COleDispatchDriver
{
public:
CBPCDeviceBase() {}// Calls COleDispatchDriver default constructor
CBPCDeviceBase(LPDISPATCH pDispatch) : COleDispatchDriver(pDispatch) {}
CBPCDeviceBase(const CBPCDeviceBase& dispatchSrc) : COleDispatchDriver(dispatchSrc)

// Attributes
public:

// Operations
public:
CString GetName();
BOOL GetHasFilename();
BOOL GetHasCA();
BOOL GetIsInput();
BOOL GetIsOutput();
BOOL GetHasChannel();
long GetStatus();
CString GetProdName();
CString GetFileName();
void SetFileName(LPCTSTR lpszNewValue);
long GetChannel();
void SetChannel(long nNewValue);
BOOL GetChannelAvailable(long nChannel);
long GetImageSourceWidth();
long GetImageSourceHeight();
long GetCurrentState();
double GetCurrentPosition();
void SetCurrentPosition(double newValue);
double GetDuration();
double GetPrerollTime();
void SetPrerollTime(double newValue);
double GetRate();
void SetRate(double newValue);
long GetCountryCode();
void SetCountryCode(long nNewValue);
long GetVideoFrequency();
long GetAudioFrequency();
long GetDefaultVideoType();
void SetDefaultVideoType(long nNewValue);
long GetDefaultAudioType();
void SetDefaultAudioType(long nNewValue);
long GetVideoSubchannel();
void SetVideoSubchannel(long nNewValue);
long GetAudioSubchannel();
void SetAudioSubchannel(long nNewValue);
void SetVolume(long nNewValue);
long GetVolume();
void SetBalance(long nNewValue);
```

```
long GetBalance();
void SetPower(BOOL bNewValue);
BOOL GetPower();
void SetOverScan(long nNewValue);
long GetOverScan();
long GetProviderRating();
BOOL GetProviderStatus();
long GetProviderEPGMask();
LPDISPATCH GetHistoryItems();
LPDISPATCH GetEmailMessages();
CString GetUserName_();
void SetUserName(LPCTSTR lpszNewValue);
CString GetUserArea();
void SetUserArea(LPCTSTR lpszNewValue);
LPDISPATCH GetItemDetails(long Priority, LPDISPATCH pInDetails);
long Command(long nCommand);
void Run();
void Pause();
void Stop();
void Refresh();
void ResetProviderSystem();
LPDISPATCH BuyItem(long Priority, LPDISPATCH pInDetails);
LPDISPATCH CancelItem(LPDISPATCH pInDetails);
void DisplayConfigDialog();
void HandleCardChaining(BOOL fOK);
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately befor

#endif // !defined(AFX_BPCDEVICEBASE_H__4E088A12_0E75_11D1_A073_00A0C9054174__INCLU
```

## BPCDEV.CPP

```
//
// bpcdev.cpp: Implements the CBPCDevices class
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//
// Machine generated IDispatch wrapper class(es) created by Microsoft Visual C++

// NOTE: Do not modify the contents of this file.  If this class is regenerated by
//  Microsoft Visual C++, your modifications will be overwritten.


#include "stdafx.h"
#include "bpcdev.h"

// Dispatch interfaces referenced by this interface
#include "BPCBase.h"
```

```
/////////////////////////////////////////////////////////////////////////////
// CBPCDevices properties

/////////////////////////////////////////////////////////////////////////////
// CBPCDevices operations

long CBPCDevices::GetCount()
{
long result;
InvokeHelper(0x5dd, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

LPUNKNOWN CBPCDevices::GetNewEnum()
{
LPUNKNOWN lpUnk = NULL;
InvokeHelper(-4, DISPATCH_PROPERTYGET, VT_UNKNOWN, (void*)&lpUnk, NULL);
return lpUnk;
}

void CBPCDevices::SetHWnd(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x5de, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

void CBPCDevices::SetLcid(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x5df, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

void CBPCDevices::SetNotify(LPDISPATCH newValue)
{
static BYTE parms[] =
VTS_DISPATCH;
InvokeHelper(0x5e0, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 newValue);
}

long CBPCDevices::GetColorKey()
{
long result;
InvokeHelper(0x5e2, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

void CBPCDevices::SetColorKey(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x5e2, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

long CBPCDevices::GetPriority()
{
long result;
```

```
InvokeHelper(0x5e3, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

void CBPCDevices::SetPriority(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x5e3, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

CBPCDeviceBase CBPCDevices::GetInput()
{
LPDISPATCH pDispatch;
InvokeHelper(0x5e4, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&pDispatch, NULL);
return CBPCDeviceBase(pDispatch);
}

void CBPCDevices::SetInput(LPDISPATCH newValue)
{
static BYTE parms[] =
VTS_DISPATCH;
InvokeHelper(0x5e4, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 newValue);
}

CBPCDeviceBase CBPCDevices::GetOutput()
{
LPDISPATCH pDispatch;
InvokeHelper(0x5e5, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&pDispatch, NULL);
return CBPCDeviceBase(pDispatch);
}

void CBPCDevices::SetOutput(LPDISPATCH newValue)
{
static BYTE parms[] =
VTS_DISPATCH;
InvokeHelper(0x5e5, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 newValue);
}

BOOL CBPCDevices::GetVideoOn()
{
BOOL result;
InvokeHelper(0x411, DISPATCH_PROPERTYGET, VT_BOOL, (void*)&result, NULL);
return result;
}

void CBPCDevices::SetVideoOn(BOOL bNewValue)
{
static BYTE parms[] =
VTS_BOOL;
InvokeHelper(0x411, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 bNewValue);
}

CBPCDeviceBase CBPCDevices::Item(const VARIANT& v)
{
LPDISPATCH pDispatch;
static BYTE parms[] =
VTS_VARIANT;
InvokeHelper(0x5dc, DISPATCH_METHOD, VT_DISPATCH, (void*)&pDispatch, parms,
&v);
return CBPCDeviceBase(pDispatch);
}

void CBPCDevices::Tune(long lTuningSpace, long Channel, long VideoSubchannel, long
{
static BYTE parms[] =
VTS_I4 VTS_I4 VTS_I4 VTS_I4;
InvokeHelper(0x5e6, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 lTuningSpace, Channel, VideoSubchannel, AudioSubchannel);
}

long CBPCDevices::TSDeviceCount(long lTuningSpace)
{
long result;
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x5e7, DISPATCH_METHOD, VT_I4, (void*)&result, parms,
lTuningSpace);
return result;
}
```

## BPCDEV.H

```
//
// bpcdev.h: Defines the CBPCDevices class
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//
#if !defined(AFX_BPCDEVICES_H__4E088A11_0E75_11D1_A073_00A0C9054174__INCLUDED_)
#define AFX_BPCDEVICES_H__4E088A11_0E75_11D1_A073_00A0C9054174__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
// Machine generated IDispatch wrapper class(es) created by Microsoft Visual C++

// NOTE: Do not modify the contents of this file.  If this class is regenerated by
//  Microsoft Visual C++, your modifications will be overwritten.


// Dispatch interfaces referenced by this interface
class CBPCDeviceBase;

/////////////////////////////////////////////////////////////////////////////
// CBPCDevices wrapper class

class CBPCDevices : public COleDispatchDriver
{
public:
CBPCDevices() {}// Calls COleDispatchDriver default constructor
CBPCDevices(LPDISPATCH pDispatch) : COleDispatchDriver(pDispatch) {}
CBPCDevices(const CBPCDevices& dispatchSrc) : COleDispatchDriver(dispatchSrc) {}

// Attributes
public:

// Operations
public:
long GetCount();
void SetHWnd(long nNewValue);
void SetLcid(long nNewValue);
void SetNotify(LPDISPATCH newValue);
long GetColorKey();
void SetColorKey(long nNewValue);
long GetPriority();
void SetPriority(long nNewValue);
CBPCDeviceBase GetInput();
void SetInput(LPDISPATCH newValue);
CBPCDeviceBase GetOutput();
void SetOutput(LPDISPATCH newValue);
BOOL GetVideoOn();
void SetVideoOn(BOOL bNewValue);
CBPCDeviceBase Item(const VARIANT& v);
void Tune(long lTuningSpace, long Channel, long VideoSubchannel, long AudioSubchann
long TSDeviceCount(long lTuningSpace);
LPUNKNOWN GetNewEnum();
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately befor

#endif // !defined(AFX_BPCDEVICES_H__4E088A11_0E75_11D1_A073_00A0C9054174__INCLUDED
```

## BPCVID.CPP

```
//
// bpcvid.cpp: Implements the CBPCVid class
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//
// Machine generated IDispatch wrapper class(es) created by Microsoft Visual C++

// NOTE: Do not modify the contents of this file.  If this class is regenerated by
//  Microsoft Visual C++, your modifications will be overwritten.


#include "stdafx.h"
#include "bpcvid.h"

// Dispatch interfaces referenced by this interface
#include "BPCBase.h"
```

```
#include "BPCDev.h"

/////////////////////////////////////////////////////////////////////////////
// CBPCVid

IMPLEMENT_DYNCREATE(CBPCVid, CWnd)

/////////////////////////////////////////////////////////////////////////////
// CBPCVid properties

/////////////////////////////////////////////////////////////////////////////
// CBPCVid operations

BOOL CBPCVid::GetPower()
{
BOOL result;
InvokeHelper(0x3f0, DISPATCH_PROPERTYGET, VT_BOOL, (void*)&result, NULL);
return result;
}

void CBPCVid::SetPower(BOOL bNewValue)
{
static BYTE parms[] =
VTS_BOOL;
InvokeHelper(0x3f0, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 bNewValue);
}

double CBPCVid::GetStartTime()
{
double result;
InvokeHelper(0x3ea, DISPATCH_PROPERTYGET, VT_R8, (void*)&result, NULL);
return result;
}

void CBPCVid::SetStartTime(double newValue)
{
static BYTE parms[] =
VTS_R8;
InvokeHelper(0x3ea, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 newValue);
}

double CBPCVid::GetStopTime()
{
double result;
InvokeHelper(0x3eb, DISPATCH_PROPERTYGET, VT_R8, (void*)&result, NULL);
return result;
}

void CBPCVid::SetStopTime(double newValue)
{
static BYTE parms[] =
VTS_R8;
InvokeHelper(0x3eb, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 newValue);
}

BOOL CBPCVid::GetVideoOn()
{
BOOL result;
InvokeHelper(0x3ec, DISPATCH_PROPERTYGET, VT_BOOL, (void*)&result, NULL);
return result;
}

void CBPCVid::SetVideoOn(BOOL bNewValue)
{
static BYTE parms[] =
VTS_BOOL;
InvokeHelper(0x3ec, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 bNewValue);
}

BOOL CBPCVid::GetClosedCaption()
{
BOOL result;
InvokeHelper(0x3ed, DISPATCH_PROPERTYGET, VT_BOOL, (void*)&result, NULL);
return result;
}

void CBPCVid::SetClosedCaption(BOOL bNewValue)
{
static BYTE parms[] =
VTS_BOOL;
InvokeHelper(0x3ed, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 bNewValue);
}

BOOL CBPCVid::GetDebug()
{
BOOL result;
InvokeHelper(0x3ee, DISPATCH_PROPERTYGET, VT_BOOL, (void*)&result, NULL);
return result;
}

void CBPCVid::SetDebug(BOOL bNewValue)
{
static BYTE parms[] =
VTS_BOOL;
InvokeHelper(0x3ee, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 bNewValue);
}

long CBPCVid::GetDeviceCount()
{
long result;
InvokeHelper(0x3e9, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

CBPCDeviceBase CBPCVid::GetInput()
{
LPDISPATCH pDispatch;
InvokeHelper(0x3fc, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&pDispatch, NULL);
return CBPCDeviceBase(pDispatch);
}

void CBPCVid::SetInput(LPDISPATCH newValue)
{
static BYTE parms[] =
VTS_DISPATCH;
InvokeHelper(0x3fc, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 newValue);
}

CBPCDeviceBase CBPCVid::GetOutput()
```

```
{
LPDISPATCH pDispatch;
InvokeHelper(0x3fd, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&pDispatch, NULL);
return CBPCDeviceBase(pDispatch);
}

void CBPCVid::SetOutput(LPDISPATCH newValue)
{
static BYTE parms[] =
VTS_DISPATCH;
InvokeHelper(0x3fd, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 newValue);
}

long CBPCVid::GetColorKey()
{
long result;
InvokeHelper(0x3f4, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

void CBPCVid::SetColorKey(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x3f4, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

CString CBPCVid::GetFileName()
{
CString result;
InvokeHelper(0x3f5, DISPATCH_PROPERTYGET, VT_BSTR, (void*)&result, NULL);
return result;
}

void CBPCVid::SetFileName(LPCTSTR lpszNewValue)
{
static BYTE parms[] =
VTS_BSTR;
InvokeHelper(0x3f5, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 lpszNewValue);
}

long CBPCVid::GetPriority()
{
long result;
InvokeHelper(0x3f6, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

void CBPCVid::SetPriority(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x3f6, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

long CBPCVid::GetVolume()
{
long result;
InvokeHelper(0x3f9, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

void CBPCVid::SetVolume(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x3f9, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

long CBPCVid::GetBalance()
{
long result;
InvokeHelper(0x3fa, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

void CBPCVid::SetBalance(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x3fa, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

long CBPCVid::GetImageSourceHeight()
{
long result;
InvokeHelper(0x3fb, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

long CBPCVid::GetImageSourceWidth()
{
long result;
InvokeHelper(0x3f2, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

short CBPCVid::GetMovieWindowSetting()
{
short result;
InvokeHelper(0x3f3, DISPATCH_PROPERTYGET, VT_I2, (void*)&result, NULL);
return result;
}

void CBPCVid::SetMovieWindowSetting(short nNewValue)
{
static BYTE parms[] =
VTS_I2;
InvokeHelper(0x3f3, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

long CBPCVid::GetCurrentState()
{
long result;
InvokeHelper(0x3fe, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

double CBPCVid::GetCurrentPosition()
```

```
{
double result;
InvokeHelper(0x3ff, DISPATCH_PROPERTYGET, VT_R8, (void*)&result, NULL);
return result;
}

void CBPCVid::SetCurrentPosition(double newValue)
{
static BYTE parms[] =
VTS_R8;
InvokeHelper(0x3ff, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 newValue);
}

double CBPCVid::GetDuration()
{
double result;
InvokeHelper(0x400, DISPATCH_PROPERTYGET, VT_R8, (void*)&result, NULL);
return result;
}

double CBPCVid::GetPrerollTime()
{
double result;
InvokeHelper(0x3f1, DISPATCH_PROPERTYGET, VT_R8, (void*)&result, NULL);
return result;
}

void CBPCVid::SetPrerollTime(double newValue)
{
static BYTE parms[] =
VTS_R8;
InvokeHelper(0x3f1, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 newValue);
}

double CBPCVid::GetRate()
{
double result;
InvokeHelper(0x402, DISPATCH_PROPERTYGET, VT_R8, (void*)&result, NULL);
return result;
}

void CBPCVid::SetRate(double newValue)
{
static BYTE parms[] =
VTS_R8;
InvokeHelper(0x402, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 newValue);
}

long CBPCVid::GetLocaleID()
{
long result;
InvokeHelper(0x403, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

void CBPCVid::SetLocaleID(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x403, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

LPDISPATCH CBPCVid::GetFont()
{
LPDISPATCH result;
InvokeHelper(DISPID_FONT, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&result, NULL);
return result;
}

void CBPCVid::SetFont(LPDISPATCH newValue)
{
static BYTE parms[] =
VTS_DISPATCH;
InvokeHelper(DISPID_FONT, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 newValue);
}

short CBPCVid::GetDisplayMode()
{
short result;
InvokeHelper(0x401, DISPATCH_PROPERTYGET, VT_I2, (void*)&result, NULL);
return result;
}

void CBPCVid::SetDisplayMode(short nNewValue)
{
static BYTE parms[] =
VTS_I2;
InvokeHelper(0x401, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

long CBPCVid::GetHWnd()
{
long result;
InvokeHelper(DISPID_HWND, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

CBPCDevices CBPCVid::GetDevices()
{
LPDISPATCH pDispatch;
InvokeHelper(0x409, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&pDispatch, NULL);
return CBPCDevices(pDispatch);
}

void CBPCVid::Run()
{
InvokeHelper(0x404, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

void CBPCVid::Pause()
{
InvokeHelper(0x405, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

void CBPCVid::Stop()
{
InvokeHelper(0x406, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

void CBPCVid::Close()
```

```
{
InvokeHelper(0x407, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

void CBPCVid::Open(LPCTSTR FileName)
{
static BYTE parms[] =
VTS_BSTR;
InvokeHelper(0x408, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 FileName);
}

void CBPCVid::Login(LPCTSTR UserName, LPCTSTR Password)
{
static BYTE parms[] =
VTS_BSTR VTS_BSTR;
InvokeHelper(0x3f8, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 UserName, Password);
}

void CBPCVid::Tune(long lTuningSpace, long Channel, long VideoSubchannel, long Audi
{
static BYTE parms[] =
VTS_I4 VTS_I4 VTS_I4 VTS_I4;
InvokeHelper(0x40b, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 lTuningSpace, Channel, VideoSubchannel, AudioSubchannel);
}

long CBPCVid::TSDeviceCount(long lTuningSpace)
{
long result;
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x40c, DISPATCH_METHOD, VT_I4, (void*)&result, parms,
lTuningSpace);
return result;
}

void CBPCVid::Refresh()
{
InvokeHelper(DISPID_REFRESH, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

void CBPCVid::AboutBox()
{
InvokeHelper(0xfffffdd8, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}
```

## BPCVID.H

```
//
// bpcvid.h: Defines the CBPCVid class
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//
#if !defined(AFX_BPCVID_H__4E088A10_0E75_11D1_A073_00A0C9054174__INCLUDED_)
#define AFX_BPCVID_H__4E088A10_0E75_11D1_A073_00A0C9054174__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
// Machine generated IDispatch wrapper class(es) created by Microsoft Visual C++

// NOTE: Do not modify the contents of this file.  If this class is regenerated by
//  Microsoft Visual C++, your modifications will be overwritten.


// Dispatch interfaces referenced by this interface
class CBPCDeviceBase;
class CBPCDevices;

/////////////////////////////////////////////////////////////////////////////
// CBPCVid wrapper class

class CBPCVid : public CWnd
{
protected:
DECLARE_DYNCREATE(CBPCVid)
public:
CLSID const& GetClsid()
{
static CLSID const clsid
= { 0x31263ec0, 0x2957, 0x11cf, { 0xa1, 0xe5, 0x0, 0xaa, 0x9e, 0xc7, 0x97, 0x0 } };
return clsid;
}
virtual BOOL Create(LPCTSTR lpszClassName,
LPCTSTR lpszWindowName, DWORD dwStyle,
const RECT& rect,
CWnd* pParentWnd, UINT nID,
CCreateContext* pContext = NULL)
{ return CreateControl(GetClsid(), lpszWindowName, dwStyle, rect, pParentWnd, nID);

    BOOL Create(LPCTSTR lpszWindowName, DWORD dwStyle,
const RECT& rect, CWnd* pParentWnd, UINT nID,
CFile* pPersist = NULL, BOOL bStorage = FALSE,
BSTR bstrLicKey = NULL)
{ return CreateControl(GetClsid(), lpszWindowName, dwStyle, rect, pParentWnd, nID,
pPersist, bStorage, bstrLicKey); }

// Attributes
public:

// Operations
public:
BOOL GetPower();
void SetPower(BOOL bNewValue);
double GetStartTime();
void SetStartTime(double newValue);
double GetStopTime();
void SetStopTime(double newValue);
BOOL GetVideoOn();
void SetVideoOn(BOOL bNewValue);
BOOL GetClosedCaption();
void SetClosedCaption(BOOL bNewValue);
```

```
BOOL GetDebug();
void SetDebug(BOOL bNewValue);
long GetDeviceCount();
CBPCDeviceBase GetInput();
void SetInput(LPDISPATCH newValue);
CBPCDeviceBase GetOutput();
void SetOutput(LPDISPATCH newValue);
long GetColorKey();
void SetColorKey(long nNewValue);
CString GetFileName();
void SetFileName(LPCTSTR lpszNewValue);
long GetPriority();
void SetPriority(long nNewValue);
long GetVolume();
void SetVolume(long nNewValue);
long GetBalance();
void SetBalance(long nNewValue);
long GetImageSourceHeight();
long GetImageSourceWidth();
short GetMovieWindowSetting();
void SetMovieWindowSetting(short nNewValue);
long GetCurrentState();
double GetCurrentPosition();
void SetCurrentPosition(double newValue);
double GetDuration();
double GetPrerollTime();
void SetPrerollTime(double newValue);
double GetRate();
void SetRate(double newValue);
long GetLocaleID();
void SetLocaleID(long nNewValue);
LPDISPATCH GetFont();
void SetFont(LPDISPATCH newValue);
short GetDisplayMode();
void SetDisplayMode(short nNewValue);
long GetHWnd();
CBPCDevices GetDevices();
void Run();
void Pause();
void Stop();
void Close();
void Open(LPCTSTR FileName);
void Login(LPCTSTR UserName, LPCTSTR Password);
void Tune(long lTuningSpace, long Channel, long VideoSubchannel, long AudioSubchann
long TSDeviceCount(long lTuningSpace);
void Refresh();
void AboutBox();
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately befor

#endif // !defined(AFX_BPCVID_H__4E088A10_0E75_11D1_A073_00A0C9054174__INCLUDED_)
```

## RESOURCE.H

```
// resource.h: Header file defining new resource IDs.
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by VdMFC.rc
//
#define IDD_VDMFC_DIALOG                102
#define IDR_MAINFRAME                   128
#define IDC_LISTBOX                     1000
#define IDC_EDITFILENAME                1004
#define IDC_SETINPUT                    1005
#define IDC_SETOUTPUT                   1006
#define IDC_VIDCNTRL                    1007
#define IDC_EDITCHANNEL                 1008
#define IDC_SETFILENAME                 1009
#define IDC_SETCHANNEL                  1010
#define IDC_VIDEOON                     1011
#define IDC_PLAY                        1012
#define IDC_PAUSE                       1013
#define IDC_STOP                        1014

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        130
#define _APS_NEXT_COMMAND_VALUE         32771
#define _APS_NEXT_CONTROL_VALUE         1006
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

## STDAFX.CPP

```
// stdafx.cpp : source file that includes just the standard includes
//VdMFC.pch will be the pre-compiled header
//stdafx.obj will contain the pre-compiled type information
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//

#include "stdafx.h"
```

## STDAFX.H

```
// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//

#if !defined(AFX_STDAFX_H__4E088A0A_0E75_11D1_A073_00A0C9054174__INCLUDED_)
#define AFX_STDAFX_H__4E088A0A_0E75_11D1_A073_00A0C9054174__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#define VC_EXTRALEAN// Exclude rarely-used stuff from Windows headers

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#include <afxdisp.h>        // MFC OLE automation classes
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>// MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT


//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately befor

#endif // !defined(AFX_STDAFX_H__4E088A0A_0E75_11D1_A073_00A0C9054174__INCLUDED_)
```

## VDMFC.CPP

```
// VdMFC.cpp : Defines the class behaviors for the application.
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//

#include "stdafx.h"
```

```
#include "VdMFC.h"
#include "VdMFCDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CVdMFCApp

BEGIN_MESSAGE_MAP(CVdMFCApp, CWinApp)
//{{AFX_MSG_MAP(CVdMFCApp)
// NOTE - the ClassWizard will add and remove mapping macros here.
//    DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG
ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CVdMFCApp construction

CVdMFCApp::CVdMFCApp()
{
// TODO: add construction code here,
// Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CVdMFCApp object

CVdMFCApp theApp;

/////////////////////////////////////////////////////////////////////////////
// CVdMFCApp initialization

BOOL CVdMFCApp::InitInstance()
{
AfxEnableControlContainer();

// Standard initialization
// If you are not using these features and wish to reduce the size
//  of your final executable, you should remove from the following
//  the specific initialization routines you do not need.

#ifdef _AFXDLL
Enable3dControls();// Call this when using MFC in a shared DLL
#else
Enable3dControlsStatic();// Call this when linking to MFC statically
#endif

CVdMFCDlg dlg;
m_pMainWnd = &dlg;
int nResponse = dlg.DoModal();
if (nResponse == IDOK)
{
// TODO: Place code here to handle when the dialog is
//  dismissed with OK
}
else if (nResponse == IDCANCEL)
{
// TODO: Place code here to handle when the dialog is
//  dismissed with Cancel
}

// Since the dialog has been closed, return FALSE so that we exit the
//  application, rather than start the application's message pump.
return FALSE;
}
```

## VDMFC.H

```
// VdMFC.h : main header file for the VDMFC application
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//

#if !defined(AFX_VDMFC_H__4E088A06_0E75_11D1_A073_00A0C9054174__INCLUDED_)
#define AFX_VDMFC_H__4E088A06_0E75_11D1_A073_00A0C9054174__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"// main symbols

/////////////////////////////////////////////////////////////////////////////
// CVdMFCApp:
// See VdMFC.cpp for the implementation of this class
//

class CVdMFCApp : public CWinApp
{
public:
CVdMFCApp();

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CVdMFCApp)
public:
virtual BOOL InitInstance();
//}}AFX_VIRTUAL

// Implementation

//{{AFX_MSG(CVdMFCApp)
// NOTE - the ClassWizard will add and remove member functions here.
//    DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};


/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately befor

#endif // !defined(AFX_VDMFC_H__4E088A06_0E75_11D1_A073_00A0C9054174__INCLUDED_)
```

## VDMFCDLG.CPP

```
//
// VdMFCDlg.cpp: Implements the CVdMFCDlg class, which is
//               the application's main dialog.
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//

#include "stdafx.h"
#include <initguid.h>
#include "VdMFC.h"
#include "VdMFCDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CVdMFCDlg dialog

CVdMFCDlg::CVdMFCDlg(CWnd* pParent /*=NULL*/)
: CDialog(CVdMFCDlg::IDD, pParent)
{
//{{AFX_DATA_INIT(CVdMFCDlg)
m_channel = 0;
m_filename = _T("");
//}}AFX_DATA_INIT
// Note that LoadIcon does not require a subsequent DestroyIcon in Win32
m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CVdMFCDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CVdMFCDlg)
DDX_Control(pDX, IDC_VIDCNTRL, m_CVid);
DDX_Text(pDX, IDC_EDITCHANNEL, m_channel);
DDX_Text(pDX, IDC_EDITFILENAME, m_filename);
//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CVdMFCDlg, CDialog)
//{{AFX_MSG_MAP(CVdMFCDlg)
ON_WM_PAINT()
ON_WM_QUERYDRAGICON()
ON_BN_CLICKED(IDC_SETINPUT, OnSetInput)
ON_BN_CLICKED(IDC_SETOUTPUT, OnSetOutput)
ON_BN_CLICKED(IDC_SETCHANNEL, OnSetChannel)
ON_BN_CLICKED(IDC_SETFILENAME, OnSetFile)
ON_BN_CLICKED(IDC_PLAY, OnPlay)
ON_BN_CLICKED(IDC_PAUSE, OnPause)
ON_BN_CLICKED(IDC_STOP, OnStop)
ON_BN_CLICKED(IDC_VIDEOON, OnVideoOn)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CVdMFCDlg message handlers

BOOL CVdMFCDlg::OnInitDialog()
{
CDialog::OnInitDialog();

// Set the icon for this dialog.  The framework does this automatically
//  when the application's main window is not a dialog
SetIcon(m_hIcon, TRUE);// Set big icon
SetIcon(m_hIcon, FALSE);// Set small icon

// Assign the list-box control to the CListBox variable and
// then verify.
VERIFY( box.SubclassDlgItem( IDC_LISTBOX, this ) );

// Obtain pointers to unknown and enumerate-variant objects.

LPUNKNOWN lpunk;
LPENUMVARIANT lpenumvar;

// Declare and allocate a Devices object and assign the
// collection of all the available input and output devices
// for the video control to this object.
CBPCDevices pDevices = m_CVid.GetDevices();

// Assign the unknown object to the value returned from the
// GetNewEnum function. GetNewEnum is a custom function added
// to the CBPCDevices class.
lpunk = pDevices.GetNewEnum();

// Query the unknown object to obtain an enumerate-variant
// object and then release the unknown object.
if( lpunk == NULL )
return FALSE;

VERIFY( SUCCEEDED( lpunk->QueryInterface( IID_IEnumVARIANT, ( void** )&lpenumvar )
lpunk->Release();

long celt;
COleVariant var;

// While there are still devices that can provide input or
// output for the video control, construct a DeviceBase object
// from the next variant obtained from the enumerate-variant
// object and add the device's name to the list box.
while( S_OK == lpenumvar->Next( 1, &var, ( unsigned long* )&celt ) )
{
ASSERT( var.vt == VT_DISPATCH );
CBPCDeviceBase id( var.pdispVal );
box.AddString( id.GetName() );

var.Clear();
}

// Release the enumerate-variant object.
lpenumvar->Release();

return TRUE;  // return TRUE  unless you set the focus to a control
}

// If you add a minimize button to your dialog, you will need the code below
//  to draw the icon.  For MFC applications using the document/view model,
//  this is automatically done for you by the framework.

void CVdMFCDlg::OnPaint()
{
if (IsIconic())
{
CPaintDC dc(this); // device context for painting

SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

// Center icon in client rectangle
int cxIcon = GetSystemMetrics(SM_CXICON);
int cyIcon = GetSystemMetrics(SM_CYICON);
CRect rect;
GetClientRect(&rect);
int x = (rect.Width() - cxIcon + 1) / 2;
int y = (rect.Height() - cyIcon + 1) / 2;

// Draw the icon
dc.DrawIcon(x, y, m_hIcon);
}
else
{
CDialog::OnPaint();
}
}

// The system calls this to obtain the cursor to display while the user drags
//  the minimized window.
HCURSOR CVdMFCDlg::OnQueryDragIcon()
{
return (HCURSOR) m_hIcon;
}

void CVdMFCDlg::OnSetInput()
{
LPUNKNOWN lpunk;
LPENUMVARIANT lpenumvar;
BOOL bFound = FALSE;
IBPCDeviceBase* pDevice;

CBPCDevices pDevices = m_CVid.GetDevices();
lpunk = pDevices.GetNewEnum(); //GetNewEnum custom function see
   //details
```

```
if( lpunk == NULL )
return;

VERIFY( SUCCEEDED( lpunk->QueryInterface( IID_IEnumVARIANT, ( void** )&lpenumvar )
lpunk->Release();

for( int i = 0; i < m_CVid.GetDeviceCount(); i++ )
{
long celt;
COleVariant var;

if( SUCCEEDED( lpenumvar->Next( 1, &var, (unsigned long*)&celt ) ) )
{
var.punkVal->QueryInterface( IID_IBPCDeviceBase, (void**)&pDevice );

if( pDevice != NULL )
{
BSTR bsDeviceName;
pDevice->get_Name( &bsDeviceName );
CString str( bsDeviceName );
SysFreeString( bsDeviceName );

CString string;
int sel = box.GetCurSel();

if( sel == LB_ERR )
{
AfxMessageBox( "Please select a device from the list" );
pDevice->Release();
lpenumvar->Release();
var.Clear();
return;
}
box.GetText( sel, string );

if( str == string )
{
m_CVid.SetInput( pDevice );
m_pDeviceBase = pDevice;
bFound = TRUE;
}
pDevice->Release();
}
var.Clear();
}
if( bFound )
break;
}
ASSERT( lpenumvar != NULL );
lpenumvar->Release();
}

void CVdMFCDlg::OnSetOutput()
{
LPUNKNOWN lpunk;
LPENUMVARIANT lpenumvar;
BOOL bFound = FALSE;
IBPCDeviceBase* pDevice;

CBPCDevices pDevices = m_CVid.GetDevices();
lpunk = pDevices.GetNewEnum(); //GetNewEnum custom function see
   //details
```

```
if( lpunk == NULL )
return;

VERIFY( SUCCEEDED( lpunk->QueryInterface( IID_IEnumVARIANT, ( void** )&lpenumvar )
lpunk->Release();

for( int i = 0; i < m_CVid.GetDeviceCount(); i++ )
{
long celt;
COleVariant var;

if( SUCCEEDED( lpenumvar->Next( 1, &var, (unsigned long*)&celt ) ) )
{
var.punkVal->QueryInterface( IID_IBPCDeviceBase, (void**)&pDevice );

if( pDevice != NULL )
{
BSTR bsDeviceName;
pDevice->get_Name( &bsDeviceName );
CString str( bsDeviceName );
SysFreeString( bsDeviceName );

CString string;
int sel = box.GetCurSel();

if( sel == LB_ERR )
{
AfxMessageBox( "Please select a device from the list" );
pDevice->Release();
lpenumvar->Release();
var.Clear();
return;
}
box.GetText( sel, string );

if( str == string )
{
m_CVid.SetOutput( pDevice );
m_pDeviceBase = pDevice;
bFound = TRUE;
}
pDevice->Release();
}
var.Clear();
}
if( bFound )
break;
}
ASSERT( lpenumvar != NULL );
lpenumvar->Release();
}

void CVdMFCDlg::OnVideoOn()
{
m_CVid.SetVideoOn( !m_CVid.GetVideoOn() );

}

void CVdMFCDlg::OnSetChannel()
{
UpdateData( TRUE );
 CBPCDeviceBase id;
id.AttachDispatch( m_pDeviceBase, FALSE );
id.SetChannel( m_channel );
}

void CVdMFCDlg::OnSetFile()
{
UpdateData( TRUE );
m_CVid.SetFileName( m_filename );

}

void CVdMFCDlg::OnPlay()
{
m_CVid.Run();

}

void CVdMFCDlg::OnPause()
{
m_CVid.Pause();

}

void CVdMFCDlg::OnStop()
{
m_CVid.Stop();

}
```

## VDMFCDLG.H

```
//
// VdMFCDlg.h: Defines the CVdMFCDlg class, which defines the
//             behavior of the application's main dialog.
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//
//{{AFX_INCLUDES()
#include "bpcvid.h"
#include "bpcbase.h"
#include "bpcdev.h"
#include "vidtypes.h"
//}}AFX_INCLUDES

#if !defined(AFX_VDMFCDLG_H__4E088A08_0E75_11D1_A073_00A0C9054174__INCLUDED_)
#define AFX_VDMFCDLG_H__4E088A08_0E75_11D1_A073_00A0C9054174__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
```

```
#endif // _MSC_VER >= 1000

/////////////////////////////////////////////////////////////////////////////
// CVdMFCDlg dialog

class CVdMFCDlg : public CDialog
{
// Construction
public:
CVdMFCDlg(CWnd* pParent = NULL);// standard constructor

// Dialog Data
//{{AFX_DATA(CVdMFCDlg)
enum { IDD = IDD_VDMFC_DIALOG };
CBPCVidm_CVid;
longm_channel;
CStringm_filename;
//}}AFX_DATA

// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CVdMFCDlg)
protected:
virtual void DoDataExchange(CDataExchange* pDX);// DDX/DDV support
//}}AFX_VIRTUAL

// Implementation
protected:
CListBox box;
CBPCDeviceBase m_id;
HICON m_hIcon;

// Generated message map functions
//{{AFX_MSG(CVdMFCDlg)
virtual BOOL OnInitDialog();
afx_msg void OnPaint();
afx_msg HCURSOR OnQueryDragIcon();
afx_msg void OnSetChannel();
afx_msg void OnPause();
afx_msg void OnVideoOn();
afx_msg void OnStop();
afx_msg void OnSetInput();
afx_msg void OnSetOutput();
afx_msg void OnPlay();
afx_msg void OnSetFile();
//}}AFX_MSG

DECLARE_MESSAGE_MAP()
private:
IBPCDeviceBase* m_pDeviceBase;
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately befor

#endif // !defined(AFX_VDMFCDLG_H__4E088A08_0E75_11D1_A073_00A0C9054174__INCLUDED_)
```

## BPCBASE.CPP

```
//
// bpcbase.cpp: Implements the CBPCDeviceBase class
//
// Copyright (C) 1997 Microsoft Corporation
// All rights reserved.
//
// This source code is only intended as a supplement to the
// Broadcast Architecture Programmer's Reference.
// For detailed information regarding Broadcast
// Architecture, see the reference.
//
// Machine generated IDispatch wrapper class(es) created by Microsoft Visual C++

// NOTE: Do not modify the contents of this file.  If this class is regenerated by
//  Microsoft Visual C++, your modifications will be overwritten.


#include "stdafx.h"
#include "bpcbase.h"


/////////////////////////////////////////////////////////////////////////////
// CBPCDeviceBase properties

/////////////////////////////////////////////////////////////////////////////
// CBPCDeviceBase operations

CString CBPCDeviceBase::GetName()
{
CString result;
InvokeHelper(0x3e9, DISPATCH_PROPERTYGET, VT_BSTR, (void*)&result, NULL);
return result;
}

BOOL CBPCDeviceBase::GetHasFilename()
{
BOOL result;
InvokeHelper(0x3ed, DISPATCH_PROPERTYGET, VT_BOOL, (void*)&result, NULL);
return result;
}

BOOL CBPCDeviceBase::GetHasCA()
{
BOOL result;
InvokeHelper(0x410, DISPATCH_PROPERTYGET, VT_BOOL, (void*)&result, NULL);
return result;
}

BOOL CBPCDeviceBase::GetIsInput()
{
BOOL result;
InvokeHelper(0x3ea, DISPATCH_PROPERTYGET, VT_BOOL, (void*)&result, NULL);
return result;
}

BOOL CBPCDeviceBase::GetIsOutput()
{
BOOL result;
InvokeHelper(0x3eb, DISPATCH_PROPERTYGET, VT_BOOL, (void*)&result, NULL);
return result;
}
```

```
BOOL CBPCDeviceBase::GetHasChannel()
{
BOOL result;
InvokeHelper(0x3ec, DISPATCH_PROPERTYGET, VT_BOOL, (void*)&result, NULL);
return result;
}

long CBPCDeviceBase::GetStatus()
{
long result;
InvokeHelper(0x3ef, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

CString CBPCDeviceBase::GetProdName()
{
CString result;
InvokeHelper(0x3f0, DISPATCH_PROPERTYGET, VT_BSTR, (void*)&result, NULL);
return result;
}

CString CBPCDeviceBase::GetFileName()
{
CString result;
InvokeHelper(0x3f1, DISPATCH_PROPERTYGET, VT_BSTR, (void*)&result, NULL);
return result;
}

void CBPCDeviceBase::SetFileName(LPCTSTR lpszNewValue)
{
static BYTE parms[] =
VTS_BSTR;
InvokeHelper(0x3f1, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 lpszNewValue);
}

long CBPCDeviceBase::GetChannel()
{
long result;
InvokeHelper(0x3f2, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

void CBPCDeviceBase::SetChannel(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x3f2, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

BOOL CBPCDeviceBase::GetChannelAvailable(long nChannel)
{
BOOL result;
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x3f5, DISPATCH_PROPERTYGET, VT_BOOL, (void*)&result, parms,
nChannel);
return result;
}

long CBPCDeviceBase::GetImageSourceWidth()
{
```

```
long result;
InvokeHelper(0x3f7, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

long CBPCDeviceBase::GetImageSourceHeight()
{
long result;
InvokeHelper(0x3f8, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

long CBPCDeviceBase::GetCurrentState()
{
long result;
InvokeHelper(0x412, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

double CBPCDeviceBase::GetCurrentPosition()
{
double result;
InvokeHelper(0x3f9, DISPATCH_PROPERTYGET, VT_R8, (void*)&result, NULL);
return result;
}

void CBPCDeviceBase::SetCurrentPosition(double newValue)
{
static BYTE parms[] =
VTS_R8;
InvokeHelper(0x3f9, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 newValue);
}

double CBPCDeviceBase::GetDuration()
{
double result;
InvokeHelper(0x3fa, DISPATCH_PROPERTYGET, VT_R8, (void*)&result, NULL);
return result;
}

double CBPCDeviceBase::GetPrerollTime()
{
double result;
InvokeHelper(0x3fb, DISPATCH_PROPERTYGET, VT_R8, (void*)&result, NULL);
return result;
}

void CBPCDeviceBase::SetPrerollTime(double newValue)
{
static BYTE parms[] =
VTS_R8;
InvokeHelper(0x3fb, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 newValue);
}

double CBPCDeviceBase::GetRate()
{
double result;
InvokeHelper(0x3fc, DISPATCH_PROPERTYGET, VT_R8, (void*)&result, NULL);
return result;
}
```

```
void CBPCDeviceBase::SetRate(double newValue)
{
static BYTE parms[] =
VTS_R8;
InvokeHelper(0x3fc, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 newValue);
}

long CBPCDeviceBase::GetCountryCode()
{
long result;
InvokeHelper(0x3fd, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

void CBPCDeviceBase::SetCountryCode(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x3fd, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

long CBPCDeviceBase::GetVideoFrequency()
{
long result;
InvokeHelper(0x3fe, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

long CBPCDeviceBase::GetAudioFrequency()
{
long result;
InvokeHelper(0x3ff, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

long CBPCDeviceBase::GetDefaultVideoType()
{
long result;
InvokeHelper(0x400, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

void CBPCDeviceBase::SetDefaultVideoType(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x400, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

long CBPCDeviceBase::GetDefaultAudioType()
{
long result;
InvokeHelper(0x401, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

void CBPCDeviceBase::SetDefaultAudioType(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x401, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

long CBPCDeviceBase::GetVideoSubchannel()
{
long result;
InvokeHelper(0x402, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

void CBPCDeviceBase::SetVideoSubchannel(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x402, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

long CBPCDeviceBase::GetAudioSubchannel()
{
long result;
InvokeHelper(0x403, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

void CBPCDeviceBase::SetAudioSubchannel(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x403, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

void CBPCDeviceBase::SetVolume(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x407, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

long CBPCDeviceBase::GetVolume()
{
long result;
InvokeHelper(0x407, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

void CBPCDeviceBase::SetBalance(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x408, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

long CBPCDeviceBase::GetBalance()
{
long result;
InvokeHelper(0x408, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}
```

```
void CBPCDeviceBase::SetPower(BOOL bNewValue)
{
static BYTE parms[] =
VTS_BOOL;
InvokeHelper(0x40b, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 bNewValue);
}

BOOL CBPCDeviceBase::GetPower()
{
BOOL result;
InvokeHelper(0x40b, DISPATCH_PROPERTYGET, VT_BOOL, (void*)&result, NULL);
return result;
}

void CBPCDeviceBase::SetOverScan(long nNewValue)
{
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x413, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 nNewValue);
}

long CBPCDeviceBase::GetOverScan()
{
long result;
InvokeHelper(0x413, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

long CBPCDeviceBase::GetProviderRating()
{
long result;
InvokeHelper(0x51d, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

BOOL CBPCDeviceBase::GetProviderStatus()
{
BOOL result;
InvokeHelper(0x51e, DISPATCH_PROPERTYGET, VT_BOOL, (void*)&result, NULL);
return result;
}

long CBPCDeviceBase::GetProviderEPGMask()
{
long result;
InvokeHelper(0x519, DISPATCH_PROPERTYGET, VT_I4, (void*)&result, NULL);
return result;
}

LPDISPATCH CBPCDeviceBase::GetHistoryItems()
{
LPDISPATCH result;
InvokeHelper(0x520, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&result, NULL);
return result;
}

LPDISPATCH CBPCDeviceBase::GetEmailMessages()
{
LPDISPATCH result;
InvokeHelper(0x521, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&result, NULL);
return result;
}

CString CBPCDeviceBase::GetUserName_()
{
CString result;
InvokeHelper(0x51b, DISPATCH_PROPERTYGET, VT_BSTR, (void*)&result, NULL);
return result;
}

void CBPCDeviceBase::SetUserName(LPCTSTR lpszNewValue)
{
static BYTE parms[] =
VTS_BSTR;
InvokeHelper(0x51b, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 lpszNewValue);
}

CString CBPCDeviceBase::GetUserArea()
{
CString result;
InvokeHelper(0x51c, DISPATCH_PROPERTYGET, VT_BSTR, (void*)&result, NULL);
return result;
}

void CBPCDeviceBase::SetUserArea(LPCTSTR lpszNewValue)
{
static BYTE parms[] =
VTS_BSTR;
InvokeHelper(0x51c, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms,
 lpszNewValue);
}

LPDISPATCH CBPCDeviceBase::GetItemDetails(long Priority, LPDISPATCH pInDetails)
{
LPDISPATCH result;
static BYTE parms[] =
VTS_I4 VTS_DISPATCH;
InvokeHelper(0x518, DISPATCH_PROPERTYGET, VT_DISPATCH, (void*)&result, parms,
Priority, pInDetails);
return result;
}

long CBPCDeviceBase::Command(long nCommand)
{
long result;
static BYTE parms[] =
VTS_I4;
InvokeHelper(0x3f4, DISPATCH_METHOD, VT_I4, (void*)&result, parms,
nCommand);
return result;
}

void CBPCDeviceBase::Run()
{
InvokeHelper(0x40c, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

void CBPCDeviceBase::Pause()
{
InvokeHelper(0x40e, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}
```

```
void CBPCDeviceBase::Stop()
{
InvokeHelper(0x40d, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

void CBPCDeviceBase::Refresh()
{
InvokeHelper(0x40f, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

void CBPCDeviceBase::ResetProviderSystem()
{
InvokeHelper(0x515, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

LPDISPATCH CBPCDeviceBase::BuyItem(long Priority, LPDISPATCH pInDetails)
{
LPDISPATCH result;
static BYTE parms[] =
VTS_I4 VTS_DISPATCH;
InvokeHelper(0x516, DISPATCH_METHOD, VT_DISPATCH, (void*)&result, parms,
Priority, pInDetails);
return result;
}

LPDISPATCH CBPCDeviceBase::CancelItem(LPDISPATCH pInDetails)
{
LPDISPATCH result;
static BYTE parms[] =
VTS_DISPATCH;
InvokeHelper(0x517, DISPATCH_METHOD, VT_DISPATCH, (void*)&result, parms,
pInDetails);
return result;
}

void CBPCDeviceBase::DisplayConfigDialog()
{
InvokeHelper(0x51a, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

void CBPCDeviceBase::HandleCardChaining(BOOL fOK)
{
static BYTE parms[] =
VTS_BOOL;
InvokeHelper(0x523, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
 fOK);
}
```

# WEBTUNE.TXT

The WebTune directory contains an HTML document that
Internet Explorer uses to display a Video ActiveX control.