-- --

# VEX

## Video Extension to X
**Version 5.8**

Todd Brunhoff

Tektronix, Inc.
Tektronix Research Laboratories

## 1. Overview

The spirit of this extension is to provide an X interface to the generally interesting aspects of displaying live video in windows, capturing graphics from windows and converting them to a video signal, and managing the network of connections to and from devices that may receive or produce these signals, such as video tape recorders. Of those people that have provided input, some disagree on the amount of functionality that should be provided in a server, particularly on control of video devices like VCRs. To satisfy all parties, the protocol is designed so that the server may advertise many or few limitations on its capabilities. For example, a server may advertise that it has an input device, but that it has no control over it and looks like a cable TV channel; alternatively, it can say that it can control playback speed, direction, seeking, etc.

For applications that need access to the digitized video pixels, this extension allows normal pixmaps or windows with core visual classes to hold video, which in turn allows read and write access to the pixels via the core protocol or other extensions such as the X Image Extension. For hardware that does not actually digitize any pixels, VEX introduces two new visual classes: **VideoColor** and **VideoGray**. Their purpose is to express the visual aspects of a visible but untouchable video picture, hence drawing on a window with one of these visual classes has no effect and the pixels are undefined.

Today, video input and output hardware has complex limitations and capabilities; the capabilities do not always scale in expected ways, nor do the limitations always make sense. The VEX extension can provide large amounts of information about these limitations and capabilities. It is intended that the client, through the use of convenience routines in the VEXlib implementation, will be able to easily determine what it can and can't do without the overhead of a round-trip request to the server.

## 2. Terminology

**Connectivity state**

At any time, the connectivity among video inputs, outputs and devices has a defined state. The **ChangeConnectivity** request may change the connectivity state by selecting a prototype matrix and making or breaking an explicit connection.

**Control atoms**

A VEX server may provide controls for various aspects of video input hardware, video output hardware and video devices. Each type of control has an **Atom** which is a unique ISO Latin-1 string interned by the server. The correspondence of strings and control types are described in a separate document, VEX Devices and Controls (not yet written).

**Full-motion video**

The display of video on the screen at a rate at least as fast as that defined by the encoded video signal. For example NTSC is defined as (roughly) 30 frames per second and 60 fields per second.

**Opaque pixels**

Some video input hardware uses a single frame buffer to hold both video pixels and graphic pixels. If the video pixels are stored in a form that does not match a core visual class, then the meaning of those pixels, from a client perspective, is said to be opaque.

**Still video**

A single, static frame acquired from a video signal.

**Video device**

An external device for producing or receiving a video signal, regardless of the video signal format. For example, a videodisc player or VCR. It is identified by a VIDEODEVICE id.

**Video digitization**

A process that converts a video image from an encoded signal into a digital form having a supported depth and format.

**Video input**

The term *video input* in general refers to an incoming signal that is decoded or digitized to produce a picture on the workstation display. In this document, video input refers to the digitizing or decoding hardware, and it is identified by a VIDEOIN id. Logically, a client will connect an output device, such as a video disk player, to a video input using the **ChangeConnectivity** request, and then use **RenderVideo** to direct in which drawable the picture will appear.

**Video input region (VIR)**

This is defined as the subset of pixels in a VW or a pixmap that is actually being updated by a **RenderVideo** request.

**Video output**

The term *video output* in general refers to signals produced by encoding workstation graphics. In this document, video output refers to the encoding hardware, and it is identified by a VIDEOOUT id. Logically, a client will connect an input device, such as a video tape recorder, to a video output using the **ChangeConnectivity** request, and then use **CaptureGraphics** to direct from which drawable the graphics will be captured.

**Video output region (VOR)**

This is defined as the set of pixels on the screen or in a pixmap that are being encoded by a **CaptureGraphics** request, whether it is accomplished by reading pixels from a frame buffer or by analog conversion.

**Video window (VW)**

A video window is defined as any window actively displaying video or one that has the potential to display nothing but video. Specifically,

- a window having a visual class **VideoColor** or **VideoGray**.
- a window with a core visual class, and actively servicing a **RenderVideo** request.

## 3. Visuals

There are three important classes of video input hardware. Each class produces a video picture on the screen using a different method, and the methods affects the visual class used to represent the window or pixmap that contains it. VEX attempts to recognize all three classes:

- Core visual pixels. Many instances of hardware choose to represent a video picture in a frame buffer that is no different than most graphic frame buffers. In this case nothing new need be added, because one of the core visual classes is sufficient.
- Inaccessible pixels. Some low-cost video input hardware is contained in a separate subsystem where either pixels are never digitized, or if they are, the pixels are available only at low serial line speeds. The analog video picture is merged with the workstation graphics analog signal such that the picture is updated very quickly. Therefore, this hardware will continue to be reasonable as long as video is not a standard option for every X display. For this, VEX introduces two new visual class, **VideoGray** and **VideoColor**, whose semantics essentially describe a write-only visual: video pixel values cannot be read. This implies that pixmap access for these visual types will never be offered.
- Opaque pixels. A few instances of hardware choose to represent digitized pixels as an encoded color difference, such as YUV; Y,R-Y,B-Y; or YIQ. This encoding is more compact than RGB, but cannot be represented with any of the core visual classes. Rather than invent more new color models and visual classes, VEX allows the use of whatever core visual class matches the frame buffer's "normal" mode, but pixels that are modified

by **RenderVideo** are opaque. This implies that while the pixels can be read, their meaning is undefined (note that a vendor may choose to provide more information about the meaning of these pixels via control mechanisms, described later). See the description for the RENDERMODEL type and the **RenderVideo** request.

Windows created with **VideoGray** display only the luminance portion of a video signal; windows created with **VideoColor** are able to display a color picture if chrominance information is available in the video signal. A side effect of this visual class is that the visible rendition of regions not covered by a **RenderVideo** request are undefined, including the window's border. However, it is expected that server implementors will pick reasonable defaults.

## 4. Types

VIDEOIN: 32-bit value (top three bits guaranteed to be zero)

VIDEOOUT: 32-bit value (top three bits guaranteed to be zero)

VIDEODEVICE: 32-bit value (top three bits guaranteed to be zero)

VIDEOIO: VIDEOIN or VIDEOUT

VIDEOID: VIDEOIN, VIDEOOUT or VIDEODEVICE

When supplied to a request, VIDEODEVICE must be one supplied by the **QueryVideo** request, else **Match** error. When a VIDEOIN or VIDEOOUT is supplied to a request, it must be one returned by the **CreateVideo** request, else **Match** error.

VREFERENCE: 32-bit value (top three bits guaranteed to be zero)

This is a special id supplied in a VIDEOGEOMETRY structure that uniquely identifies the corresponding resource.

TRANSITION: [ source, destination: VIDEOID
matrix1, matrix2: CARD32 ]

The TRANSITION type is used by the **QueryVideo** request to describe a transition on the directed graph of adjacency matrices (returned by **QueryVideo**) when the given source and destination are connected or disconnected. The matrix1 and matrix2 element are indices into the list of adjacency matrices; the matrices are numbered from 0. The two matrices they index describe the possible connectivity among video inputs, outputs, and devices before and after the source and destination are connected or disconnected.

CONTROL: [ type: {Device, Input, Output}
name: ATOM
id: VIDEODEVICE or VREFERENCE
setting-format: { 8, 16, 32 }
setting-length: CARD32
description-format: { 8, 16, 32 }
description-length: CARD32
description-data: LISTofINT8 or LISTofINT16 or LISTofINT32 ]

The CONTROL structure is used to represent the capabilities of the "knobs" and "dials" on various video hardware. It is returned by the **QueryVideo** request. The name in a control is a control atom interned by the extension when it is initialized. The id element identifies which resource the control affects. For video devices, the type is Device and the id is a VIDEODEVICE; For video inputs and outputs, the type is Input or Output and the id is a VREFERENCE. The setting format and length describe the structure expected in a **ChangeVideoControls** request. The setting length is the number of bytes expected in the value element of a SETTING structure whose id and name match those in the control; if the length is zero, then SETTINGS for this control may have any length. The description length is the number of bytes in the description data field. The setting format and description format describes the arrangement of data cells so the server may properly swap a SETTING before interpreting it, and a CONTROL before sending it, respectively. The meaning of the list of values in a SETTING, the meaning of the description data in a CONTROL, and the appropriate names for controls are described in another document, VEX Devices and Controls (not yet written).

SETTING: [ name: ATOM
id: VIDEOID
format: { 8, 16, 32 }
value: LISTofINT8 or LISTofINT16 or LISTofINT32 ]

This type is used to change the setting of a named control associated with a named VIDEOID. Only the **ChangeVideoControls** and **QueryVideoControls** requests utilize it. The control to be changed is named by the atom and is associated with the named id. If the length advertised in the corresponding CONTROL is not zero, then the value must have the same length in bytes, else **Length** error.

FRACTION: [ numerator, denominator: INT32 ]

The FRACTION type is used to represent a rational number; the denominator must be strictly positive (> 0) in all requests, else **Value** error.

FRACTIONRANGE: [ num-base, num-inc, num-limit: INT32
num-type: {**Linear**, **Geometric**}
denom-base, denom-inc, denom-limit: INT32
denom-type: {**Linear**, **Geometric**}]

A FRACTIONRANGE is used in a PLACEMENT, described below. The set of fractions represented by FRACTIONRANGE depend on the types, and for each type, the range of numerators and denominators of each fraction are as follows:

**Linear**: $n$ : {$n$ = base to limit step inc

**Geometric**: $n^{exp}$ : {$n$ = base; exp = inc to limit step 1

A FRACTIONRANGE supplied by **QueryVideo** is guaranteed never to generate a zero denominator. For **Linear**, the limit is always guaranteed to be *base* plus some integral multiple of the increment. For example, a FRACTIONRANGE of {512, 128, 1024, **Linear**, 512, 512, 1024, **Linear**} creates a set containing

$$\left\{\frac{512}{512}, \frac{640}{512}, \frac{768}{512}, \frac{896}{512}, \frac{1024}{512}, \frac{512}{1024}, \frac{640}{1024}, \frac{768}{1024}, \frac{896}{1024}, \frac{1024}{1024}\right\}.$$

A FRACTIONRANGE of {2, 0, 2, **Geometric**, 1, 1, 127, **Linear**} creates the set

$$\left\{\left(\frac{2^0}{1} = \right)\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{127}, \frac{2}{2}, \frac{2}{3}, \ldots, \frac{2}{127}, \frac{4}{2}, \frac{4}{3}, \ldots, \frac{4}{127}\right\}.$$

RECTANGLERANGE: [ base, limit: RECTANGLE
x-inc, y-inc: INT16
width-inc, height-inc: CARD16
type: {**Linear**, **Geometric**}]

The RECTANGLERANGE is ultimately used in the VIDEOGEOMETRY type below. It describes the possible geometries of sources and destinations for **RenderVideo** and **CaptureGraphics** requests. The set of rectangles described by a single RECTANGLERANGE are all combinations of four sets derived for x, y, width and height. The members of the sets are determined based on the type as follows.

| | | |
|---|---|---|
| **Linear**: | x | = { base.x, base.x + x-inc, base.x + 2·x-inc, ..., limit.x } |
| | y | = { base.y, base.y + y-inc, base.y + 2·y-inc, ..., limit.y } |
| | width | = { base.width, base.width + width-inc, base.width + 2·width-inc, ..., limit.width } |
| | height | = { base.height, base.height + height-inc, base.height + 2·height-inc, ..., limit.height } |
| **Geometric**: | x | = { $\text{x-inc}^{\text{base.x}}$, $\text{x-inc}^{\text{base.x+1}}$, ..., $\text{x-inc}^{\text{limit.x}}$ } |
| | y | = { $\text{y-inc}^{\text{base.y}}$, $\text{y-inc}^{\text{base.y+1}}$, ..., $\text{y-inc}^{\text{limit.y}}$ } |
| | width | = { $\text{width-inc}^{\text{base.width}}$, $\text{width-inc}^{\text{base.width+1}}$, ..., $\text{width-inc}^{\text{limit.width}}$ } |
| | height | = { $\text{height-inc}^{\text{base.height}}$, $\text{height-inc}^{\text{base.height+1}}$, ..., $\text{height-inc}^{\text{limit.height}}$ } |

For example, the RECTANGLERANGE whose value is

[ base = {0, 0, 320, 240}, limit = {960, 784, 320, 240}
x-inc = 16, y-inc = 1
width-inc = 0, height-inc = 0
type = **Linear** ]

would describe a total of 47,885 {x, y, width, height} sets

{0, 0, 320, 240}
{16, 0, 320, 240}
...
{960, 0, 320, 240}
{0, 1, 320, 240}
{16, 1, 320, 240}
...
{960, 1, 320, 240}
etc.
{960, 784, 320, 240}

PLACEMENT: [ frame-rate: FRACTION }
source, destination: RECTANGLERANGE
x-scale, y-scale: FRACTIONRANGE
identity-aspect: BOOL ]

The PLACEMENT type is used in the VIDEOGEOMETRY type described below. Each placement describes the possible source and destination rectangles used in **RenderVideo** and **CaptureGraphics** requests, but restricted to the scale factors described by the x-scale and y-scale elements. For video input, the frame-rate element describes the maximum nominal frame rate that the hardware can achieve in a **RenderVideo** request using a source and destination rectangle derived from the other elements. For video output, the frame-rate element describes the maximum nominal rate at which frames can be encoded with a **CaptureGraphics** request. It is important to stress that this is only a nominal rate and may vary due to other factors such as hardware contention.

The x-scale and y-scale FRACTIONRANGE specifies the possible ratios of
destination x pixels : source x pixels
destination y pixels : source y pixels
where the ratio is numerator (destination) divided by denominator (source). The source and destination rectangles derived must have width and height ratios described by x-scale and y-scale. The identity-aspect is true if the ratio of x-scale to y-scale is 1:1; i.e. they must be equal. For example, given a PLACEMENT of

| | | |
|---|---|---|
| frame-rate | = | [ 30, 1 ] |
| source | = | [ base = {0, 0, 10, 15}, limit = {639, 479, 640, 480}<br>x-inc = 1, y-inc = 1<br>width-inc = 1, height-inc = 1<br>type = **Linear** ] |
| destination | = | [ base = {0, 0, 320, 240}, limit = {960, 784, 320, 240}<br>x-inc = 16, y-inc = 1<br>width-inc = 0, height-inc = 0<br>type = **Linear** ] |
| x-scale | = | [ base = {0, 1}, increment = {2, 1}<br>limit = 6<br>type = **Geometric** ] |
| y-scale | = | [ base = {0, 1}, increment = {2, 1}<br>limit = 5<br>type = **Geometric** ] |
| identity-aspect | = | False |

it implies that the source rectangles may be any with the constraints that

$$0 \le x \le 639$$

$$0 \le y \le 479$$

$$10 \le \text{width} \le 640$$

$$15 \le \text{height} \le 480$$

and the destination rectangles may be any selected from the earlier example. The combined rectangles are limited to those source-destination pairs where the fractions

$$\text{width-scale} = \frac{\text{destination width}}{\text{source width}} \quad \text{and} \quad \text{height-scale} = \frac{\text{destination height}}{\text{source height}}$$

can be reduced to one of the fractions from the respective sets

*width-scale* = [ {1,1}, {2,1}, {4,1}, {8,1}, {16,1}, {32,1}, {64,1} ]
*height-scale* = [ {1,1}, {2,1}, {4,1}, {8,1}, {16,1}, {32,1} ]

And for this example, using any of these source-destination pairs, the hardware can maintain a nominal 30 fps.

VIDEOGEOMETRY: [ signal-frame-rate: FRACTION
signal-field-rate: FRACTION
signal-width, signal-height: CARD16
concurrent-use: CARD16
reference-id: VREFERENCE
placement: LISTofPLACEMENT ]

This type describes the attributes of video inputs and video outputs and is returned by the **QueryVideo** request. The signal-frame-rate and signal-field-rate describe the number of frames per second and fields per second in the signal format. For input, the rate describes the signal received; for output, the signal produced.

For video input, the signal-width and signal-height describe the dimensions of the signal picture as if it were placed directly on the workstation screen. For video output, they describe the dimensions of a screen region whose pixels would map one-to-one to the output signal. It is important to note that the frame-buffer holding the video picture, if any, may not actually have these dimensions; the numbers describe the extent of the source rectangle for video input, or the destination rectangle for the video output.

For a video input and output, the concurrent-use indicates the number of times the resource may be in simultaneous use, and implies the number identifiers a client may create using **CreateVideo**. The server should publish a number that the hardware can support with respect to connectivity and quality of the input or output. Clients should use this number in deciding how many resources they may use, and a video resource manager should use this in deciding how to handle allocation among multiple clients.

The reference id uniquely identifies the resource, and this id is used by the **CreateVideo** request to create video input and output identifiers. If the same reference id appears in a VIDEOGEOMETRY returned from different screens, then the concurrent use for will be the same and a video manager can infer that one physical resource services both screens. This implies that usage of this resource on one screen reduces the available usage on another screen.

Each placement element in the list describes its own range of values. The complete range of placement parameters is derived from the entire list, but parts of one element in the list may not be combined with another. For example, source and destination rectangles described in one placement may not be combined with scale factors described in another placement.

RENDERMODEL: [ depth: CARD8
visual-id: VISUALID
opaque: BOOL
red-mask, green-mask, blue-mask: CARD32 ]

This type is used by **QueryVideo** to describe the depth and visual class of a potential source for **CaptureGraphics** or destination for **RenderVideo**. The visual id is one returned by the connection setup or the **QueryVideo** request. The boolean, *opaque*, is true when a video input is used to render to a drawable created with this visual id, and the pixel values created cannot be interpreted by a core visual class. The red, green and blue mask are those used for **CaptureGraphics** when the source is a pixmap and the colormap is **None**; and for **RenderVideo** when the destination is a pixmap and *opaque* is false.

Note that if *opaque* is true, then the visual id is guaranteed to have a core visual class.

OWNER: [ wid: WINDOW
vid: VIDEOID ]

This type is used by the **ChangeOwnership** request to establish ownership for any of the video resources: input, output and devices. The specified window identifies the client, and the videoid indicates the resource of which it gains ownership.

## 5. Errors

VEX introduces one new error type.

| Error | Description |
|---|---|
| Video | A value for a VIDEOID argument does not name a defined VIDEOIO, a VREFERENCE or VIDEODEVICE provided by QueryVideo. |

## 6. Requests

### QueryVideo

*wid*: WINDOW
==>
*screen*: CARD8
*video-depths*: LISTofDEPTHS
*allowed-depths*: LISTofRENDERMODEL
*vdev*: LISTofVIDEODEVICE
*inattr*: LISTofVIDEOGEOMETRY
*outattr*: LISTofVIDEOGEOMETRY
*video-input-models*: LISTofBITMASK {**Pixmap**, **Window**}
*video-output-models*: LISTofBITMASK {**Pixmap**, **Window**, **Composite**}
*videoid-controls*: LISTofCONTROL
*clip-size*: LISTofRECTANGLE
*input-overlap*: BOOL
*capture-overlap*: BOOL
*io-overlap*: BOOL
*adjacency-matrices*: LISTofLISTofCARD8 {**Connected**, **NotConnected**, **NExclusive**}
*transitions*: LISTofTRANSITION
*time*: TIMESTAMP

Errors: **Window**, **Match**

This request returns information about video hardware for the screen associated with the specified window. The screen is returned as the first element in the reply. If there is no video hardware available for the screen, a **Match** error is returned.

The *video-depths* specify what depths/visuals are unique for video input and output. These are guaranteed to be different from those returned in the X connection setup; the list may be null; pixmaps are supported for each depth listed. Further, the presence of a VEX extension in a server may cause the LISTofFORMAT provided by the connection setup to be extended with additional formats that would allow **GetImage** and **PutImage** access to windows or pixmaps created with depths and visuals not published in the connection setup. The definition of DEPTH is included in the core protocol's description of the connection information.

Visual ids found in *video-depths* having a core visual class (**PseudoColor**, **TrueColor**, etc.) imply that graphic requests with corresponding windows are expensive. Thus, it may be that pixels must undergo software translation before or after graphic requests. Depths and visual ids listed in *allowed-depths* that are selected from the X connection setup imply that they are not expensive, even though they may also be used for **RenderVideo** and **CaptureGraphics** requests.

The *allowed-depths* specify what depths and visual ids are supported for use with video input and video output. These depths and visual ids include those listed in video-depths plus appropriate ones selected from those provided by the X connection setup. The list is in no particular order.

The vdev lists the online devices that provide signals to a video input or accept signals from a video output. The list may be null and implies that the server has no control over any devices that may be attached to the video inputs or video outputs. One or more device ids may also appear in **QueryVideo** information for other screens and implies that the same device is available for both screens; ownership and connectivity for these devices affect availability on both screens. If a device goes offline or a new device comes online, then a **VideoChange** event will be generated.

The inattr list the attributes of the decoder or digitizer used by the video inputs. The outattr list the attributes of the encoder used by the video outputs. The entries in both lists correspond one-to-one with the unique video input renderers and video output encoders; the length of these lists imply the number of rows in video input models and video output models. **CreateVideo** and the videoid controls described in this request reference these resources by this implied index, starting from 0.

The list of video input models should be interpreted as a two dimensional array, column moving fastest and represents the relationship between the video input resources and the set of depths and visualids used to create video windows (VW). The columns are labeled left-to-right with the list of *allowed-depths*, and the rows are labeled top-to-bottom with the list of video inputs. Each cell is a bitmask containing zero or more true bits.

**Window** is asserted if a **RenderVideo** request may specify a window with the intersecting depth/visual as a destination and the intersecting video input as the source. If **Pixmap** is asserted at the intersection of an *allowed-depth* and a video input, then a **RenderVideo** request may specify a pixmap with that depth as a destination and that video input as the source (see **RenderVideo** for a discussion of pixmap pixel values). There is guaranteed to be at least one **Window** or **Pixmap** assertion in every row. In addition, there is guaranteed to be no more than one **Pixmap** assertion for every unique depth, per row: this avoids ambiguity for the RENDERMODEL use of *opaque* and the red-, green- and blue-masks.

If a window is created with a visual id whose type is **VideoGray** or **VideoColor**, then the window's pixels are always undefined. A **RenderVideo** will not change window's pixel values (although the picture may still be visible).

For example, the following array means that a VW for video input #1 must be depth 12, **TrueColor**; or depth 8, **VideoColor**; pixmaps of depth 24 can be used as a destination for a **RenderVideo** request. The #2 video input can only support pixmaps of depth 12.

| | | **VideoColor**<br>depth 8 | **TrueColor**<br>depth 12 | **None**<br>depth 24 |
|---|---|---|---|---|
| *Video in* | #1 | **Window** | **Window** | **Pixmap** |
| | #2 | 0 | **Pixmap** | 0 |

The list of video-output-models should be interpreted as a two dimensional array, column moving fastest, and represents the relationship between the video output resources and the set of depths and visual ids that each may capture. The columns are labeled left-to-right with the list of *allowed depths*, and the rows are labeled top-to-bottom with the list of video outputs. Each cell is a bitmask containing zero or more true bits.

If **Window** is asserted, then a video output can capture the intersecting depth/visual in a window, including the border. If **Composite** is asserted, and the BLEND extension is present, and a window having the intersecting depth/visual is specified in a **CaptureGraphics** request, and the window is a member of a blend triple, then the window's contents will be captured after its primaries are blended. If **Composite** is asserted, then **Window** will be asserted for the same cell; The assertion of **Composite** is guaranteed to agree with the information returned by QueryBlend. If **Composite** is not asserted, but **Window** is asserted, then capture from a window having the intersecting depth/visual will always consist of unblended pixels from the source window and its descendants of the same depth, whether or not the window is a member of a blend triple. If a descendant is a member of a blend triple, the result is undefined: it is hardware dependent. If **Pixmap** is asserted, then a pixmap with the intersecting depth can be captured (see **CaptureGraphics** description of the interpretation of pixmap pixel values). There is guaranteed to be at least one **Window** or **Pixmap** assertion in every row. In addition, there is guaranteed to be no more than one **Pixmap** assertion for every unique depth, per row: this avoids ambiguity for the use of the red-, green- and blue-masks.

For example, if the video hardware has two video outputs, one able to capture depth 12, **TrueColor**, blended windows, and one able to capture depth 24 pixmaps, then the array would look like:

| | | **TrueColor** depth 12 | **TrueColor** depth 24 |
|---|---|---|---|
| *Video out* | #1 | **Composite**, **Window** | 0 |
| | #2 | 0 | **Pixmap** |

The *videoid-controls* describes the controls, if any, for each VIDEOID. The order of the list will, in general, match the *inattr*, *outattr* and vdev, although the number of controls for any one particular VIDEOID is arbitrary. The list may be null.

The clip-size list corresponds to the list of video inputs (*inattr*), and describes the smallest rectangle that can be used to clip a VIR partly occluded by other windows. If either the width or the height are 0, then the entire VIR must be displayed and cannot be clipped. This is meant as advisory for clients and window managers as they choose window placement.

If *input-overlap* is false, then VIRs may not overlap; true otherwise. *Capture-overlap* is false if the hardware does not allow VORs to overlap; true otherwise. *Io-overlap* element is false if the hardware does not allow a VIR and a VOR to overlap; true otherwise. Note that io-overlap may be true, but video-output-models determines if a VIR may be captured or not.

In all cases, if clip-size, input-overlap, or io-overlap constraints are violated, the content of the violated regions is undefined; if capture-overlap or io-overlap is violated, then the output signal for violated regions are undefined and is hardware dependent; no errors are returned to the offending request, but a **VideoViolation** event may be generated. In addition, other constraints that VEX cannot express may be violated, and a **VideoViolation** event may be generated. Requests that may violate constraints are **RenderVideo**, **CaptureGraphics**, **MapWindow**, **UnmapWindow**, **MapSubwindows**, **UnmapSubwindows**, **ConfigureWindow**, **CirculateWindow**, **DestroyWindow**, DestroySubwindows, **ReparentWindow**.

The adjacency-matrices should be interpreted as a list of two dimensional arrays, each array the same size, each with column moving fastest. Each array represents a possible set of connections among video inputs, outputs and devices.

The columns in each individual matrix represent sources, and are labeled left-to-right with the video devices followed by the video outputs; the rows represent destinations, and are labeled top-to-bottom with the video devices followed by the video inputs. This implies that no video input may be a source, and no video output may be a destination. Each cell is made up of two CARD8 values described below. Each row is

$$(\text{devices}+\text{outputs})\cdot 2$$

CARD8 values in length and there are

$$(\text{devices}+\text{inputs})$$

rows. The value pairs in an array are **Connected** (255) if the intersecting source and destination are connected; **NotConnected** (0), otherwise. In either case both values will be the same.

Potential connectivity is represented by **NExclusive**, which is a value from 1 to 254. In this case, both values will be some **NExclusive**; the first CARD8 designates how many connections may be made for the devices or inputs in that row, and the second CARD8 designates the number of connections that may be made for the devices or outputs in that column. Any row that has, say, **5Exclusive** in its row designator, then all other row designators will have one of {**Connected**, **NotConnected**, **5Exclusive**}. Likewise, if any column has **2Exclusive** in its column designator, then all other column designators will have one of {**Connected**, **NotConnected**, **2Exclusive**}. Note that any **Connected** markings in a row or column are counted against the **NExclusive**.

The list of arrays (matrices) comprise an unordered list of nodes on a directed graph. Taken together, they describe the potential connectivity among video inputs, outputs and devices.

The transitions list provides the edges to the graph. Each entry describes a single transition between two nodes given the connection of a source and destination. The nodes are adjacency matrices in the list, indexed by the matrix1 and matrix2 elements of the TRANSITION type; the matrices are numbered from 0. Each edge described is bi-directional; i.e., disconnecting the same source and destination while in the matrix2 will transition back to matrix1. Also, other connections or disconnections may be implied by a single transition, and these edges will also appear in the transitions list. Note that since each matrix may represent one of several states, some transitions may have the same index for matrix1 and matrix2.

For example, if **QueryVideo** returns two matrices and six transitions

| | | | *sources* | |
|---|---|---|---|---|
| | *matrix 0* | dev #1 | dev #2 | out #1 |
| | dev #1 | N | row: E1<br>col: E2 | row: E1<br>col: E2 |
| *dest.* | dev #2 | N | N | C |
| | in #1 | N | C | N |

| | | | *sources* | |
|---|---|---|---|---|
| | *matrix 1* | dev #1 | dev #2 | out #1 |
| | dev #1 | N | N | N |
| *dest.* | dev #2 | row: E1<br>col: E2 | N | N |
| | in #1 | row: E1<br>col: E2 | N | N |

C: connected
N: not connected
E*n*: *n* exclusive

{ dev2, dev1, 0, 0 }
{ out1, dev1, 0, 0 }
{ dev2, in1, 1, 0 }
{ out1, dev2, 1, 0 }
{ dev1, dev2, 1, 1 }
{ dev1, in1, 1, 1 }

This states that device 2 or out 1 may be connected to device 1, but not both. If out 1 is connected to device 2, then device 2 is also connected to input 1; and if one is disconnected, then both are disconnected. Device 1 may be connected to device 2 or input 1, or both.

Note that the matrices and transitions by themselves are not enough to determine the current state. A client must select **ConnectivityNotify** for the screen; the response from the server will be a series of **VideoConnectivityState** events that describe the transitions from an initial state to the current state of the connectivity. The initial state is defined as matrix 0 with all **NExclusive** cells set to 0, or **NotConnected**. As events are delivered, a matrix that defines the state of connectivity can be calculated as described in the **VideoConnectivityState** event.

It is mathematically possible to enumerate a set of transitions that would produce a state not described by any matrix. The set of matrices and transitions returned by **QueryVideo** are guaranteed to preclude this situation.

The time element can be used to prevent race conditions between the delivery of a **VideoChange** event and the use of some other VEX requests. The value of the timestamp is constant until there is a change in the list of video devices, and is then replaced with the current server time.

**CreateVideo**

*id*: VIDEOIO
*reference*: VREFERENCE

Errors: **Alloc**, **IDChoice**, **Video**

This request creates the specified identifier handle for the video input or output resource associated with *reference*. This resource is freed on connection close for the creating client. The number of identifiers created for a single client corresponding to a particular input or output is not limited to the concurrent-use for that resource as supplied by **QueryVideo**. However, the number of ids in use at once is limited by the concurrent use limit and ownership.

**RenderVideo**

*source*: VIDEOIN
*destination*: DRAWABLE
*src-x, src-y, dest-x, dest-y*: INT16
*src-width, src-height, dest-width, dest-height*: CARD16
*full-motion*: BOOL
*priority*: CARD8
*subwindow-mode*: {**ClipByChildren**, **IncludeInferiors**}

Errors: **Drawable**, **Match**, **Value**, **Video**

The destination must be a valid drawable id, else **Drawable** error. The specified region of the source video signal is decoded and directed to the destination. The visible portion of the destination rectangle becomes a VIR.

In general, it is intended that when a window of visual class **VideoGray** or **VideoColor** is the destination, the pixels of the video picture are inaccessible.

When a window is created with a visual id from a RENDERMODEL whose *opaque* member is true, then the intent is to say that digitized pixels are accessible, but they are not defined (hardware-specific information about them may be available from a CONTROL for that video input). When the destination is a window having a core visual class, the intent is that the video pixels are fully accessible, but happen to change "very frequently". The same is true for a pixmap depth (note that there is only one RENDERMODEL per depth; see **QueryVideo**).

If the destination is a window, then **Window** must be asserted in the *video-input-models* returned by **QueryVideo** for the intersecting visual/depth of the window and the specified video input, else **Match** error. If the window is visible, then the visible pixels are displayed. If the destination is an unmapped window, then the window's pixels may be rendered, depending on the value of backing-store for the window.

If the destination is a pixmap, then **Pixmap** must be asserted in the video-input-models for the intersecting *allowed-depth* of the pixmap and the specified video input, else **Match** error. In this case, if *opaque* is false, then the red-, green- and blue-mask define the meaning of the bits after a frame has been rendered. The source and destination must belong to the same screen, else **Match** error. If *opaque* is true, then the pixels are modified, but are undefined (a vendor-specific CONTROL may provide more information about pixel meaning).

The src-x, src-y, src-width and src-height describe the extent of the source rectangle in the video picture. The dest-x, dest-y, dest-width and dest-height describe the rectangle in the destination drawable to receive the video picture. The source video picture will be scaled by

$$x\ \text{scale} = \frac{\text{dest--width}}{\text{src--width}} \qquad y\ \text{scale} = \frac{\text{dest--height}}{\text{src--height}}$$

If the source and destination rectangles and scaling factors do not match one of those provided in a single PLACEMENT for the specified video input source, a **VideoViolation** event may be generated. If the destination rectangle is valid, but some or all of it is outside the extent of the destination drawable, the result is clipped to a granularity advertised by **QueryVideo**, and a **VideoViolation** event may be generated. If the source rectangle is valid, but some or all of it is outside the extent of the video picture, the result in the corresponding region of the destination is undefined: it is hardware dependent.

The full-motion element determines whether the video input signal is continually updated or not. If it is true, then the server will try to achieve the maximum nominal rate advertised by the **QueryVideo** request for the specified scale factors and placement. If it is False, then a single still video frame will be rendered, as if the **RenderVideo** request was immediately followed with a **StopVideo** request specifying the same drawable. However, a full-motion element of False is an aid to the server to avoid unnecessary digitization.

Priority (0 to 100 inclusive, else **Value** error), represents the percentage of video bandwidth that the server should try to preserve. A value of 0 tells the server that graphic output to the screen should have top priority and that the video decoding may occasionally miss some incoming or outgoing frames. A value of 100 gives video decoding top priority meaning that the graphics performance may be degraded. It is advisory only and does not guarantee anything.

Subwindow-mode controls the treatment of the window's children. For **ClipByChildren**, the destination is clipped by all viewable children. For **IncludeInferiors**, viewable inferiors of the destination that overlap the VIR are updated with pixels from the video picture if the destination and its inferiors have a matching visual/depth; otherwise, the result in the inferiors is undefined.

If the destination window is clipped by siblings or children, a **VideoViolation** event may be generated if an overlap or clipping constraint is violated; see **QueryVideo**.

If a client has ownership of a video input, a **RenderVideo** request cannot be stopped except by an explicit **StopVideo**, a change in the mapped state of the destination drawable, a change in the size of the destination window with bitgravity ForgetGravity, destruction of the destination drawable, or a **VideoViolation** event with state **Fail**. If a client does not have ownership, or ownership is relinquished, then the request is also subject to being stopped under two other circumstances. In either case, a **VideoOverride** event will be generated.

- A **RenderVideo** request using a window W already specified in another active **RenderVideo** request, has the same effect as issuing a **StopVideo** request with arguments W and **Render**, followed by the new **RenderVideo** request.

- A **RenderVideo** request using a video input V whose concurrent use is exhausted has the same effect as issuing a **StopVideo** request with arguments W' and **Render**, where W' is the window from the earlier request, followed by the new **RenderVideo** request.

Clients that do not need access to digitized pixels should use a window having a **VideoGray** or **VideoColor** visual class as a destination. This allows the server to display the video picture using the most efficient means. With all visual classes, there is no guarantee that the signal is actually digitized.

If pixels are accessible (**Composite** set for the intersecting video input and render model; see **QueryVideo**), then **CopyArea**, **CopyPlane** and **GetImage** requests with a source region within a VIR are guaranteed to return a single frame with respect to the video signal. Note that the digitized picture may come from a signal whose frame is composed of fields, and that there may be temporal differences between fields. More device-specific information about the format of the picture may be available from one of the video input's controls.

If the specified window changes size while the request is still active, the destination x and y the original request will be adjusted to follow the bit gravity of the destination, but the width, height and scale are not adjusted. If the resulting rectangle is beyond the extent of the destination, the result is clipped appropriately; a **VideoViolation** event may be generated. In the case of ForgetGravity, a window size change will cause an implied **StopVideo** with and action of **Render** to be issued on the resized window.

Graphics may be drawn on a VW, but if the window is servicing an active **RenderVideo** request, the results are undefined.

**CaptureGraphics**

*source*: DRAWABLE
*destination*: VIDEOOUT
*cmap*: COLORMAP or **None**
*src-x*, *src-y*, *dest-x*, *dest-y*: INT16
*src-width*, *src-height*, *dest-width*, *dest-height*: CARD16
*full-motion*: BOOL
*priority*: CARD8
*subwindow-mode*: {**ClipByChildren**, **IncludeInferiors**}

Errors: **Drawable**, **Match**, **Value**, **Video**, **Colormap**

The specified region of the source is encoded into a video signal by the destination encoder. If the source is a window, then **Window** must be asserted in the video-output-models returned by **QueryVideo** for the intersecting visual/depth of the window and the specified video output, else **Match** error. If the source is a pixmap, then **Pixmap** must be asserted in the video-output-models for the intersecting depth of the pixmap and the specified video output, else **Match** error.

If the source is a window, *cmap* must be **None**, else **Match** error. If the source is a pixmap and the colormap is not **None**, then the visual id of the RENDERMODEL that intersects with the specified video output must be defined (else **Match** error), and the colormap defines the visual and colormap for the encoded picture. If the source is a pixmap and the colormap is **None**, then the interpretation of the pixel values is provided by the red-, green- and blue-mask for the RENDERMODEL. The source and destination must belong to the same screen, else **Match** error.

The src-x, src-y, src-width and src-height describe the extent of the source rectangle of the region captured. The dest-x, dest-y, dest-width and dest-height describe the rectangle in the destination video picture to receive the graphics. The source graphics region will be scaled by

$$\text{x scale} = \frac{\text{dest--width}}{\text{src--width}} \qquad \text{y scale} = \frac{\text{dest--height}}{\text{src--height}}$$

If the scaling factors applied do not match one of those provided by the **QueryVideo** request for the specified video output destination, a **VideoViolation** event may be generated. If the destination rectangle is valid, but some or all of it is outside the extent of the video picture, those pixels are clipped in the output video picture. If the source rectangle is valid, but some or all of it is beyond the extent of the source drawable, the result in the encoded video signal is undefined: it is hardware dependent.

The full-motion element determines whether the video output signal is constantly updated or not. If it is true, then the server will try to achieve the maximum rate advertised by the **QueryVideo** request for the specified scale factor and placement. If it is False, then a single still video frame will be captured. It is just as if the **CaptureGraphics** request was immediately followed with a **StopVideo** request specifying the same drawable; however, a full-motion element of False is an aid to the server to avoid extra encoding.

Priority is a value (0 to 100 inclusive, else **Value** error), representing the percentage of video bandwidth that the server should try to preserve. A value of 0 tells the server that graphic output to the screen should have top priority and that the video capture process may occasionally miss some incoming or outgoing frames. A value of 100 gives video capture top priority meaning that the graphics performance may be degraded. It is advisory only and does not guarantee anything.

Subwindow-mode controls the capture of the window's children. If **ClipByChildren** is specified, the source is clipped by all viewable children only if **Composite** is not asserted in the video-output-models for the video output destination. If **Composite** is asserted, subwindow-mode will be effectively **IncludeInferiors**, regardless of its value.

If **IncludeInferiors** is specified for the subwindow-mode, then each visible region of an inferior of the source that overlaps the VIR may captured, but only if one or both of the following are true:

- The inferior is the same depth, visual and window type as the source window.
- **Composite** is asserted in the video-output-models for the video output destination.

Otherwise, the source rectangle is clipped.

If the source rectangle is clipped by siblings or children, the appearance of the clipped regions in the encoded picture are undefined: it is hardware dependent.

If a client has ownership of a video output, a **CaptureGraphics** request cannot be stopped except by an explicit **StopVideo**, a change in the mapped state of the source drawable, a change in the size of the source window with bitgravity ForgetGravity, destruction of the source drawable, or a **VideoViolation** event with state **Fail**. If a client does not have ownership, or ownership is relinquished, then the request is also subject to being stopped under two other circumstances. In either case, a **VideoOverride** event will be generated.

- A **CaptureGraphics** request using a window W already allocated to an earlier **CaptureGraphics** request has the same effect as issuing a **StopVideo** request with arguments W and **Capture**, followed by the new **CaptureGraphics** request.
- A **CaptureGraphics** request using a video output V whose concurrent use is exhausted has the same effect as issuing a **StopVideo** request with arguments W' and **Capture**, where W' is the window from the earlier request, followed by the new **CaptureGraphics** request.

A **CaptureGraphics** request using a video output V whose concurrent use is exhausted has the same effect as issuing a **StopVideo** request with arguments W' and **Capture**, where W' is the window from the earlier request, followed by the new **CaptureGraphics** request.

If the specified window changes size while the request is still active, a **VideoViolation** event may be generated. Only the source x and y of the original request will be adjusted to follow the bit gravity of the source. If a portion of the new rectangle is beyond the extent of the source, the result in the output signal is undefined. In the case of ForgetGravity, a window size change will cause an implied **StopVideo** with an action of **Capture** to be issued on the resized window.

**StopVideo**

*idlist*: LISTofDRAWABLE
*action*: BITMASK {**Render**, **Capture**}

Errors: **Drawable**, **Value**

Each id must be a valid drawable, else **Drawable** error. If more than one id is in error, which one is reported is arbitrary.

The currently active renders and/or captures are stopped on all of the listed ids, even if one of the ids produces an error. If **Render** is specified, then **RenderVideo** requests on the listed ids are terminated. If **Capture**, is specified, then **CaptureGraphics** requests on the listed ids are terminated. Both may be specified. If neither **RenderVideo** or **CaptureGraphics** are currently active on a id, no error is reported.

If a **RenderVideo** on window W having a visual of type **VideoColor** or **VideoGray** is followed by a **StopVideo** on W, then the visible result on the screen is undefined, because it is hardware dependent. If a **RenderVideo** on a window W having a core visual class is followed by a **StopVideo** on W, the result in W is the pixels take on the value of the last frame acquired: they can be manipulated by other X requests.

In all cases, this request will generate a **VideoOverride** event for each drawable that is servicing an active **RenderVideo** or **CaptureGraphics** request.

**QueryVideoControls**

*vid*: VIDEOID
*control-list*: LISTofATOMS
==>
LISTofSETTING

Errors: **Match**, **Atom**, **Video**

This request provides the current setting associated with each of the listed controls for the listed video id. The control list must be taken from the set of control names supplied by **QueryVideo**, else **Atom** error.

If the vid or any of the control-list generate an error, no settings are returned.

**ChangeVideoControls**

*setting*: LISTofSETTING

Errors: **Match**, **Atom**, **Value**, **Video**, **Length**

This request changes the setting of each control named in each SETTING structure. The control named by the atom and associated with the specified id is set according to the value of the data. A **VideoControl** event is generated on the specified id when the control setting has actually been accomplished, failed, or was denied.

All valid settings take effect, even if some of the settings are invalid. If an error is returned, which setting generated the error is not specified.

**ChangeConnectivity**

*screen*: CARD8
*time*: TIMESTAMP
*matrix*: CARD32
*source*, *destination*: VIDEOID
*connect*: BOOL

Errors: **Match**, **Value**, **Access**, **Video**

The time specified must be later than the time returned by the last **VideoChange** event, and later than the last **VideoConnectivityState**, else **Value** error.

The *matrix* must be a valid index into the list of matrices returned by **QueryVideo**, else **Value** error. The *matrix*, *source* and *destination* must also appear in some TRANSITION as *matrix2*, *source* and *destination*, respectively, else **Match** error.

The connectivity state is changed such that all cells marked **Connected** or **NotConnected** in *matrix* are copied to the current state, and cells marked NExlcusive keep their previous value. Finally, the cell indexed by the source and destination is changed to **Connected** if *connect* is true, **NotConnected** otherwise. If the request would violate an ownership, an **Access** error is returned.

**ChangeOwnership**

*owners*: LISTofOWNER

Errors: **Window**, **Video**, **Match**

*Note:* this request is designed for the use of a video resource manager, not a client. Conventions should be developed such that a client designates interest in video resources, perhaps through properties, and the video manager allocates ownership as appropriate.

For each member of the list, the client that created the specified window is given ownership of the specified resource; if the client no longer exists, no action is taken. Owership for a video input or video output means that subsequent **RenderVideo** or **CaptureGraphics** requests are guaranteed not to be redirected, respectively. In addition, all controls associated with the resource may only be changed by the owning client. The number of clients given ownership of a single video input or output must not exceed the concurrent use limit for that resource as supplied by **QueryVideo**, else **Match** error. Note that a VIDEOIO specified in a OWNER is simply an alias for a VREFERENCE; a VIDEOIO created by any client may be used to specify the resource; and the window is used to designate the client-owner.

If a **RenderVideo** or **CaptureGraphics** request is issued for a resource, and one or more instances of the resource are in use by a client that does not have ownership, then a **StopVideo** request will be implied as described by **RenderVideo** and **CaptureGraphics**, regardless of whether the last issuing client has ownership for that resource or not. A **VideoOverride** event will be generated on the drawable. Under the same circumstances, if the issuing client does not have ownership, and all instances of the resource have ownership, the request will not take effect and a **VideoRequest** event will be generated with *redirect* set to TRUE.

Similarly, for a video device, the client indicated by the specified window is given ownership of the specified device. **ChangeConnectivity** requests are guaranteed not to violate the connectivity of that device, i.e. whatever it is immediately connected to will remain connected. Second level connectivity is not guaranteed. In addition, all **ChangeVideoControls** requests on a device will only succeed for the owning client. If a **ChangeVideoControls** request would violate ownership, a **VideoControl** event will be generated on the resource.

If a device is not in use, or a video input or output is in use for less than its specified concurrent use limit, then a request requiring that resource will always succeed, regardless of ownership. Conversely, if a device is in use or a resource is utilized to its concurrent use limit, and the client utilizing the resource does not have ownership, and a client having ownership issues a request that requires the resource, then the owning client takes precedence, and a **VideoOverride** or **VideoConnectivityState** event will be generated, as appropriate.

Non ownership is achieved by designating the root window as the client-owner or by destroying the previously designated window. If an input, output or device is not owned by a specific client, all clients may affect the device's connectivity or controls. The number of clients given ownership of a single device must never exceed one, else **Match** error.

**SelectVideoEvents**

*id*: VIDEOID or DRAWABLE
*mask*: BITMASK
{**VideoRedirect**, **ConnectivityNotify**, **SyncNotify**,
**ControlNotify**, **OverrideNotify**, **ViolationNotify**}

Errors: **Drawable**, **Video**, **Access**, **Value**, **Alloc**

This requests selects events for the specified resource. Four of the masks are for drawables: **VideoRedirect**, ConnectNotify, **OverrideNotify** and **ViolationNotify**; the **SyncNotify** mask is for a VIDEOIO; and the **ControlNotify** mask is for a VIDEOID. If any of the drawable masks are included with any of the VIDEOID masks, a **Value** error occurs. If the id is not a drawable for any of the drawable masks, a **Drawable** error occurs. If the id is not a VIDEOID for **ControlNotify**, or a VIDEOIO for **SyncNotify**, a **Video** error occurs.

Selecting **VideoRedirect** will allow a client to receive **VideoRequest** events generated on the screen of the specified drawable; only one client may select the event on a given screen, else **Access** error. After a client has selected the event, to maintain control, the client must assert **VideoRedirect** in every subsequent **SelectVideoEvents** request that specifies a drawable; the first request by the client specifying a drawable on the same screen without **VideoRedirect** asserted will release control. This selection is intented for use by a video resource manager.

Selecting **ConnectivityNotify** will allow a client to receive **VideoConnectivityState** events generated on the screen of the specified drawable. After a client has selected the event, every **SelectVideoEvents** request that specifies a drawable from the same screen must select **ConnectivityNotify**, or the event will be de-selected. For each screen, when a client selects **ConnectivityNotify** anew, a series of **VideoConnectivityState** events will be generated that will describe the transitions from an initial state that are necessary to reach the current state. The initial state, as described by **QueryVideo** is matrix 0 with all **NExclusive** cells marked as **NotConnected**.

Selecting **SyncNotify** allows the client to receive **VideoSync** events generated on the specified VideoIO. Note that VideoDevices do not generate **VideoSync** events.

Selecting **ControlNotify** allows the client to receive **VideoControl** events generated on the specified VideoID.

Selecting **OverrideNotify** will allow client to receive **VideoOverride** events generated on the named drawable.

Selecting **ViolationNotify** will allow the client to receive **VideoViolation** events generated on the named drawable.

If any of the bits generates an error, which error is returned is not specified, and the state of selected events does not change.

## 7. Core Requests

**CreateWindow**

< this is a core protocol request >

If the VEX extension is present, this request will understand two new visual classes: **VideoGray** and **VideoColor**. These are used to create one instance of a video window (VW). Visuals specified in a **CreateWindow** request must be one supported by the screen, and may be supplied by **QueryVideo**.

## 8. Events

Every VEX event also contains the least-significant 16 bits of the sequence number of the last request issued by the client that was (or is currently being) processed by the server.

**VideoConnectivityState**

*screen*: CARD8
*change*: {**Matrix0**, **Connected**, **NotConnected**}
*source*, *destination*: VIDEOID
*matrix*: CARD32
*time*: TIMESTAMP
*sequence*: CARD32
*count*: CARD16

This event is reported to the client selecting **ConnectivityNotify** on the specified screen when the connectivity of video inputs, outputs and devices changes state. The *matrix* element is an index into the list of adjacency matrices returned by **QueryVideo**.

The *change* element describes the new state of the source and destination: connected or not connected. The value **Matrix0** is a special flag that signals a return to the initial state, as described in **QueryVideo**; in this case, the *matrix*, *source* and *destination* have undefined values.

The source and destination specify two VIDEOIDs whose connectivity changes to the value specified by *change*.

The time reported is the time of the last network change caused by a **ChangeConnectivity** request.

The delivery of the **VideoConnectivityState** may not happen until some time after a **ChangeConnectivity** request, implying that the sequence number will not necessarily correspond to the originating request. The sequence number will be filled in with the sequence number of the originating request for the issuing client; for all other clients, it will be set to zero.

For a given connectivity change causing **VideoConnectivityState** events, the set of events for a given screen are guaranteed to be reported contiguously. If count is zero, then no more **VideoConnectivityState** events for this screen follow. If count is nonzero, then at least that many more **VideoConnectivityState** events for this screen follow (and possibly more).

A matrix of the actual connectivity can be maintained by the client in the following way. For each event,

- If *change* is **Matrix0**, then begin with a copy of matrix 0; set all **NExclusive** cells set to **NotConnected**.
- If a **VideoConnectivityState** event specifies a *matrix* different than the current state, then every cell marked **Connected** or **NotConnected** in the new matrix is copied, and every cell marked **NExclusive** keeps its value.
- Mark the cell indexed by the source and destination with the specified *change*: **Connected** or **NotConnected**.

**VideoChange**

*screen*: CARD8
*vdev*: VIDEODEVICE
*state*: {**Online**, **Offline**}
*time*: TIMESTAMP

This event is reported unconditionally to any client that has issued a **QueryVideo** request on the specified screen. It is generated when the named video device has changed state; i.e. a new device comes online or an existing device goes offline. If the same device is available on more than one screen, events are generated on each screen. Issuing a **QueryVideo** request will yield a new device list and adjacency matrices. New video devices are guaranteed not to have the same id as devices that were once listed, except for devices that go off line and return later: these devices will retain their previous id. This event is guaranteed to be delivered before any **VideoConnectivityState** events are delivered for the new connectivity state.

The time is what would be reported by the **QueryVideo** request.

**VideoControl**

*vid*: VIDEOID
*name*: ATOM
*state*: {**Success**, **Fail**, **Denied**}
*time*: TIMESTAMP

This event is reported to any client that has selected **ControlNotify** on the resource when a **ChangeVideoControls** request accomplishes a setting for the specified id and atom, or when it failed in an unexpected way, or if the setting violated ownership of the resource. The vid and name are those in the SETTING given to the **ChangeVideoControls** request. The state is **Success** if the setting was accomplished, **Fail** if the setting failed, or **Denied** if the setting would have violated ownership.

The time is the current server time.

**VideoViolation**

*id*: DRAWABLE
*vid*: VIDEOIO
*action*: ACTIONMASK
where ACTIONMASK is {**Scale**, **Placement**, **Clip**, **Overlap**}
*state*: {**Success**, **Subset**, **Fail**}
*time*: TIMESTAMP

This event is reported to the client selecting **ViolationNotify** on the drawable. The event is generated for the specified drawable and video input or output when scaling, placement, clipping and overlap constraints are violated. If state is **Success**, then the listed violations caused no change in the display or video output for the specified drawable, although some other drawable may be affected; If state is **Subset**, then some subset of the unobscured regions are still being displayed or captured; if the state is **Fail**, then the video input is no longer being displayed, or the capture is no longer occurring. Note that for pixmaps, **Overlap** will never be asserted.

The time is the current server time.

**VideoSync**

*id*: DRAWABLE
*vid*: VIDEOIO
*state*: {**Acquired**, **Lost**}
*time*: TIMESTAMP

This event is reported to the client selecting **SyncNotify** on the vid.

It is generated following a **RenderVideo** request with the specified id and video input, whenever the sync is lost or gained. When selected, at least one **VideoSync** event is guaranteed following a **RenderVideo** request, if one complete frame has been rendered before a **StopVideo** request is issued.

The event is also generated following a **CaptureGraphics** request with the specified id and video output, whenever the output signal begins to be produced. When selected, at least one **VideoSync** event is guaranteed following a **CaptureGraphics** request, if one complete frame has been generated before a **StopVideo** request is issued.

The time is the current server time.

**VideoOverride**

*id*: DRAWABLE
*vid*: VIDEOIO
*time*: TIMESTAMP

This event is reported to the client selecting **OverrideNotify** on the id. It is generated following a **RenderVideo** or **CaptureGraphics** request that results in an implicit **StopVideo** on the specified id; when an explicit **StopVideo** cancels an active **RenderVideo** or **CaptureGraphics**; when the drawable specified in the request destroyed or its mapped state changes; and when a **RenderVideo** or **CaptureGraphics** request completes if *full-motion* was specified as false.

The time is the current server time.

**VideoRequest**

*request*: {**Render**, **Capture**}
*redirect*: BOOL
*source*: VIDEOIN or DRAWABLE
*destination*: VIDEOOUT or DRAWABLE
*cmap*: COLORMAP or **None**
*src-x*, *src-y*, *dest-x*, *dest-y*: INT16
*src-width*, *src-height*, *dest-width*, *dest-height*: CARD16
*full-motion*: BOOL
*priority*: CARD8
*subwindow-mode*: {**ClipByChildren**, **IncludeInferiors**}
*time*: TIMESTAMP

This event is reported to the client selecting **VideoRedirect**. The request element indicates whether the request was **RenderVideo** or **CaptureGraphics**. Every **RenderVideo** or **CaptureGraphics** request using a resource for which the requesting client does not have ownership, and the concurrent use limit has been reached, and there are no non-owners using the resource, will generate a **VideoRequest** event with redirect set to true. Events are generated for requests that do not match this criteria with redirect set to false. This event is intended to be used by video resource managers.

The time is the current server time.