VEX

Devices and Controls
Version 1.3

Todd Brunhoff

Tektronix, Inc.
Tektronix Research Laboratories

1. Overview

This document is intended to define conventions for the usage and semantics of the control mechanism defined by the VEX protocol. The controls in VEX were designed to support an arbitrary set of functions on an unknown number of devices, and consequently, the descriptions in the protocol provide no semantics; this document describes those semantics. The controls defined here are those that are common among all servers supporting VEX. Some are required, some optional.

The protocol and encoding define the format of the control and setting. The VEX library document describes the C-language function definitions, structure definitions, and defined constants used as values for the structure members. This document will name and describe

- the strings that match the atom names interned by the server for each control
- the structures for the description data in each control
- the structures for the value data in each setting
- the structures for the value data for the XVEXControlEvent event(s) (new in the protocol)
- the defined constants for the above structure members
- the function definitions for the C-language library interface

Note that the descriptions here will be in terms of C structures found in the VEX Controls library interface. Most of these structures match the protocol encoding one-for-one. However, some settings are similar to the the value list used by CreateWindow: the library structure declares all the elements, but a bit-mask determines which of the structure elements are transmitted to the server. Encoding for each control is listed in the appendix.

2. Library Support

The VEX control mechanism is an extensible one. It provides a static set of data that can provide general information about a control; it provides a complete interface for reading and writing an arbitrary size "state" structure; and it allows events to be generated for each control with sixteen of the event bytes unique to each control,

The actual protocol mechanism requires that a QueryVideo request be performed to give the client the information needed to work with VEX, and included in this block of data is a list of all controls for all devices, video inputs and video outputs. The C interface for this request is `XVEXQueryVideo()`, but as described in the VEXlib interface, it isn't necessary to call it explicitly, because every VEXlib library function will automatically does it, such that all the programmer needs to do is use the portions of VEXlib that pertains to the task.

Every control consists of four parts:

- Some overall information that identifies that control. Within X, atoms are used as 32-bit unique names and VEX uses one for each control. These Atoms are "interned" by the server from ISO-LATIN1 strings, and for the standard VEX controls, which are the ones described in this document, they all begin with the characters "VEX_", such as VEX_PLAY and VEX_STOP. Other parts of the overall information are to which device the control applies, the size and format of the description data, and the size and format of the setting.
- Some static data called the control's "description data". There may be a lot or a little or no data, and the intent is to aid the client program in interpreting a control, or choosing the right values to set. For example, the VEX_SPEED control's description data contains a list of valid speeds which a video tape play can attain.
- A structure used to read and write the state of the control, called a "setting". Typically, this is a small amount of data, such as a mode word, but may be any size that the core X protocol can support. Because of byte swapping considerations on a network, the control-specific data in each setting must be equal sized elements (32-bit, 16-bit or 8-bit), although the interface described here represents the data with reasonable data types. There are two VEXlib requests that read and write raw settings, `XVEXQueryControls()` and `XVEXChangeControls()`, but this document describes an interface that is much simpler, giving the programmer a functional interface to each control.
- An event, called XVEXControlEvent by VEXlib, that informs the client whether a value

written to a control was successful or not; for example, whether the frame number chosen by VEX_FRAME is on the media. One of these events is generated every time a control is set, but more may be generated if the control defines it that way. Remember that generated events are not delivered unless selected using XVEXSelectEvents(). The event contains the device id, the atom name, a time stamp and the success/fail indicator. In addition, there are sixteen bytes that the control may use; for example, the current SMPTE time codes may be stored here.

In VEXlib, video inputs, video outputs and devices are represented by protocol identifiers, a 32-bit value with the type VideoIn, VideoOut and VideoDevice, respectively. For a VideoDevice, these ids are a member of the XVEXVdev structure. For a VideoIn and VideoOut, the identifiers must be created from a reference id in the XVEXVin and XVEXVout structure. All three of these structures contain a pointer to a list of controls for each; each control is described by a XVEXControl structure.

Also found in each of the XVEXVin, XVEXVout and XVEXVdev structures is an array of boolean variables named `control_available` that assert whether a corresponding control is available for the video resource. It is indexed by a set of constants whose name exactly matches the string used to intern the atom. For example, a (simple-minded) portion of a client that starts a video tape playback might look like:

```
XVEXVdev   *pDevs;

pDevs = XVEXGetVdevs( dpy, DefaultScreen(dpy), &ndevices);
if (ndevices < 1)
     return;

if (pDevs[0].control_available[ VEX_PLAY ])
     XVEXPlay(dpy, pDev->id);
```

Note that atoms and settings and controls are not visible, and it is the intent of this interface to hide these details. The use of indices is a convenience within the library, and the constants do not carry across the protocol. In fact, it is quite likely that there will be other controls which VEXlib knows nothing about, either because the server implements a newer version of this document and hence may offer new "VEX_..." controls, or because there are some vendor-specific controls. For this situation, and for generality, VEXlib allows the programmer it ask about availability of a control by using the strings used to intern the atoms, and the following functions

Bool XVEXIdHasControl(*display*, *vid*, *control_name*)
    Display **display*;
    VideoID   *vid*;
    char *control_name;

Bool XVEXVdevHasControl(*display*, *vdev*, *control_name*)
    Display **display*;
    XVEXVdev   *vdev*;
    char *control_name;

Bool XVEXVinHasControl(*display*, *vin*, *control_name*)
    Display **display*;
    XVEXVin   *vin*;
    char *control_name;

Bool XVEXVoutHasControl(*display*, *vout*, *control_name*)
    Display **display*;
    XVEXVout   *vout*;
    char *control_name;

display Specifies the connection to the X server.

vid Specifies the VideoID to check for the control.

vdev Specifies the pointer to the XVEXVdev to check for the control.

vin Specifies the pointer to the XVEXVin to check for the control.

vout Specifies the pointer to the XVEXVout to check for the control.

control_name
Specifies a pointer to a ISO-LATIN1 string that represents the atom name for the control.

`XVEXIdHasControl()` returns true if the specified identifier has the named control. `XVEXVdevHasControl()`, `XVEXVinHasControl()` and `XVEXVoutHasControl()` return true if the specified XVEXVdev, XVEXVin or XVEXVout has the named control. The latter three perform essentially the same test as `XVEXIdHasControl()`, but use a slightly different method to refer to the video resource.

So, another way to do the earlier example could be

```
XVEXVdev *pDev;

if (XVEXVdevHasControl(dpy, pDev, "VEX_PLAY")
     XVEXPlay(dpy, pDev->id);
```

or using an identifier

```
if (XVEXIdHasControl(dpy, pDev->id, "VEX_PLAY")
     XVEXPlay(dpy, id);
```

Writing a new value into a control requires a unique pair of identifiers: the video resource id, and the control atom. Each time a new value is written, an event of type XVEXControlEvent is generated, and when it is received by a client program, the programmer must examine this unique pair to identify which control on which video resource generated the event. The following functions provide a general interface for matching atoms with their names:

int XVEXAtomToIndex(*display*, *atom*)
    Display **display*;
    Atom *atom*;

int XVEXStringToIndex(*display*, *control_name*)
    Display **display*;
    char **control_name*;

Atom XVEXIndexToAtom(*display*, *index*)
    Display **display*;
    int *index*;

Atom XVEXStringToAtom(*display*, *control_name*)
    Display **display*;
    char **control_name*;

display Specifies the connection to the X server.

atom Specifies the atom name of a VEX control to convert to an index.

control_name
Specifies a pointer to a ISO-LATIN1 string that represents the atom name for the control.

index Specifies the index of a VEX control to convert to an atom.

`XVEXAtomToIndex()` converts the specified atom to a constant whose name is spelled precisely like string used to intern the atom; for example, VEX_PRODUCT and VEX_PLAY. `XVEXStringToIndex()` converts the specified string to the same kind of constant. `XVEXIndexToAtom()` converts an index into a VEX control atom and `XVEXStringToAtom()` converts a string to an index. Note that there may be controls (and thus, atoms) in the server that the library does not know about. For this situation, there is a special constant returned by `XVEXAtomToIndex()`, named OTHER_VEX_CONTROLS; e.g., the atom interned by the string _TEK_DVI_SWITCHES would be converted to the constant OTHER_VEX_CONTROLS by `XVEXAtomToIndex()`.

All VEX functions, including the four described above, perform index-to-atom and atom-to-index conversion using an internal per-display cache of atom names and the corresponding string. For this reason, only VEX-specific atoms will work with the library.

If a simpler interface is desired, each of the control descriptions in the next section includes a function to verify that an XVEXControlEvent corresponds to an identifier-atom pair. For example, checking for an event may look like:

```
VideoID          id;
XVEXControlEvent event;
...
XVEXJogFrames(dpy, id, direction);
XIfEvent(dpy, (XEvent *)&event, XVEXIsJogEvent, id);
return (event.state == VEXControlSuccess); /* true if jog succeeded */
```

The implementation for XVEXIsJogEvent() shown below is fairly simple, enough so that it could be moved to a user program, if the software design required it. Note that XVEXIsJogEvent() is designed to be compatible with the Xlib routines XIfEvent(), XCheckIfEvent() and XPeekIfEvent().

```
Bool XVEXIsJogEvent(dpy, event, id)
     Display          *dpy;
     XVEXControlEvent *event;
     VideoID          id;
{
     if (event->vid == id
      && event->type == XVEXBaseEvent(dpy) + VEXControl
      && event->name == XVEXIndexToAtom(dpy, VEX_JOG))
          return True;
     else
          return False;
}
```

3. Control Descriptions

In the descriptions that follow, the capitalized name that heads each section is the string used to intern the atom for this control, and is also a defined constant for use with VEXlib. For example, if the VEX extension returns a control named VEX_STOP, a client may intern the ISO-Latin-1 string "VEX_STOP" to receive an atom whose value is the same as that in the control (see the core protocol description for InternAtom and the Xlib description for the XGetAtomName and XInternAtom functions). Actually, this practice is discouraged for VEX applications because a cache of the atoms is kept in the VEXlib library, as described in the previous section.

Every control is different. Some have decription data in the control and others do not. Some controls can be written with a setting, others are read only, others support both, and a few support neither. In the sections that follow, each control name will be followed by

- an overview of the control and general use
- a structure describing the control's description data, if applicable
- a structure describing the control's setting value, if applicable
- a detailed description of the structures
- the C-language bindings for the library interface to utilize the features
- a description of the programming aspects for the library

The library interfaces described here make use of structures that represent controls and their associated settings. Control structures are actually returned by the `XVEXQueryVideo()` library function, but are typically obtained with the VEX library routines XVEXGetVins(), XVEXGetVouts() and XVEXGetVdevs(). Each of these routines returns a pointer to structure that holds, among other things, a pointer to a XVEXControl structure. When any of the paragraphs below mention "a control's description data", that data is stored in one of these XVEXControl structures, and can be extracted with one of the library routines associated with that control. Depending on the kind of data, it may be allocated by the library, and the user program must free it.

Most of the controls can be read or written, at the protocol level, using a so-called setting. Settings are available at the library level in XVEXSetting structures that can be obtained with through the function XVEXCreateSettingList(). However, the library interfaces described here create settings on the fly because it is more efficient, and XVEXSetting structures are never used. XVEXSetting structures may be removed in future versions.

3.1. Function Arguments

The method employed by the library functions usually requires the programmer to pass three or more arguments:

- a pointer to the display structure
- a video resource identifier, found in the XVEXVdev structure or created with the XVEXCreateVideo() function.
- a pointer to a structure particular to the control, such as a timecode or a segment list or a

speed; this is what the document refers to as "a setting's value field"

It should be noted that the C-language structures described in this section do not exactly match the size of data that is actually transmitted to the server; they are designed to support a programming style consistent with Xlib and VEXlib. The precise encoding for all controls and settings is found at the end of the document.

3.2. Function Return Values

All of the functions described in this document return one of three things:

- Status. Like Xlib, a zero status means failure and non-zero status means success. Failure can mean any of several different things, such as no VEX extension in the server, an error reply when reading a control, or failure to map a VideoID to a XVEXControl structure. Also like Xlib, if you get a failure, you probably need to start debugging. Frequently a failure is the result of invalid arguments.
- A Bool (True or False). In all cases, true is a positive confirmation that not only did the software avoid all errors, but the condition being asked about is True. False, on the other hand, can mean the queried condition is false, as well as an internal library failure which are usually due to invalid arguments.
- A structure pointer. Most controls that can be read have a fixed sized structure associated with the returned data, and the value is returned through an argument passed in to the reading routine which has a status return. Controls that return an arbitrary length of data have routines that return a pointer to the returned data, and in all cases, the pointer and all of its associated data is allocated with a single Xmalloc(), and should be freed with a single XFree(). A NULL return value can mean allocation failure or any other error mentioned above.

3.3. Classes of Controls

Controls can be grouped loosely into three classes:

Players and Recorders

There are many video recorders and players currently available, each offering different capabilities and features. There are several controls marked in the table below for such a device. The controls address the common set of features and controls that can be used in a device independent fashion. Features unique to a device are not addressed in this document.

Video Inputs and Outputs

Many of the capabilities offered by video input and output hardware are expressed in the VEX protocol, but some aspects, such as horizontal adjustments, brightness, filters, and other adjustments that do not fit in a graphic-minded system, must be handled by controls. A few of the controls address some of those requirements.

Pseudo Controls

A few controls are meant to control the controls or give global information. For example, a future control will allow a system administrator to reconfigure the set of devices controlled by a server without shutting down the server.

The following table summarizes the complete list of controls.

	video input	video output	device	has description	is readable	is writable
VEX_ADDRESS_MODE			√		√	√
VEX_BLANK_SEARCH			√		√	
VEX_DISPLAY_MODE			√		√	√
VEX_FIELD_MODE	√	√			√	√
VEX_FILL_COLOR	√	√			√	√
VEX_FRAME			√	√	√	√
VEX_JOG			√			√
VEX_OVERLAY			√	√	√	√
VEX_PLAY			√	√		√
VEX_PRODUCT	√	√	√	√		
VEX_PROTECT			√	√	√	
VEX_RECORD			√			√
VEX_ROLL			√	√	√	√
VEX_SEGMENT			√	√	√	√
VEX_SPEED			√	√	√	√
VEX_STOP			√			√
VEX_TIMECODE			√	√	√	√
VEX_VIDEO_ADJUSTMENT	√			√	√	√

3.4. Interaction Among Controls

Some of the controls work in concert, each depending on information provided by one or more other controls. VEX_SEGMENT depends on VEX_ADDRESS_MODE, VEX_SPEED, and either VEX_FRAME or VEX_TIMECODE. For some of the controls, if you find one associated with a video resource, you will find others as well.

[more information needed]

VEX_ADDRESS_MODE

The VEX_ADDRESS_MODE control allows a client to select the method of addressing video media: either with frames or time codes.

```
int   mode;
```

There is no description field for this control. The `mode` element describes the structure of setting's value field.

`mode`

This specifies whether other controls will address video media using frames or using time codes. If its value is `VEXFrameAddress`, then all addressing is done with frames as defined by VEX_FRAME. If the value is `VEXTimecodeAddress`, then all addressing is done with time codes as defined by VEX_TIMECODE. For example, when the mode is frames, a VEX_SEGMENT setting will define the segment in terms of the frame specified by VEX_FRAME.

The value `VEXNoAddress` is a special case for unaddressable media such a Constant Linear Velocity video discs (CLV); this value cannot be set, but will be returned when such a media is in the device.

Time codes and frame numbers are not always the same, nor can one always be calculated from the other. NTSC, for example, is actually 59.94 fields per second, meaning that over one hundred seconds there will be *not* be 6000 fields or 3000 frames; there will only be 2997 frames, thus three time codes must be dropped during that time. The Society for Motion Picture and Television Engineers (SMPTE) has defined a standard and predictable method for dropping frames: two frames every minute, on the minute, but not on the ten minutes.

Unfortunately, not everyone adheres to standards, and practically speaking, mistakes or problems can occur. For example,

- A tape may be prepared in a way that time codes are not monotonically increasing. For example, there may be several places on a single tape with time code 01:14:02:12:1 (1 hour, 14 minutes, 2 seconds, frame 12, first field).
- A frame number can be encoded in line 17 of the picture and may have no relation to the time code encoded for the same frame.
- Some devices, such as video disc players, deal only with frame numbers, while tape players deal primarily with time codes.

For this reason, this document describes both frames and time codes. The combination of devices, controllers and the server implementor will determine which of these controls are available for a device. This implies that for a particular device, VEX_FRAME may be offered, or VEX_TIMECODE, or both or neither.

Status XVEXSetAddressMode(*display*, *vid*, *mode*)
    Display **display*;
    VideoID *vid*;
    int *mode*;

Status XVEXGetAddressMode(*display*, *vid*, *mode_return*)
    Display **display*;
    VideoID *vid*;
    int **mode_return*;

display Specifies the connection to the X server.

vid Specifies the VideoID having the control to read or write.

mode Specifies the mode for the frame addressing.

mode_return Returns the mode for the frame addressing.

The function `XVEXSetAddressMode()` changes the current address mode. The function `XVEXGetAddressMode()` returns the current address mode.

VEX_BLANK_SEARCH

The VEX_BLANK_SEARCH control allows a client to obtain a list of unrecorded sections on write-once video media. It returns a complete list of all unrecorded segments in the form of start and stop frame numbers. If this control is present for a device, a client should use it before beginning to record.

```
typedef struct {
      int  start;
      int  stop;
} XVEXBlank;
```

There is no description field for this control. The `XVEXBlank` is the structure of setting's value field. This control is read-only, so attempting to write the control with `XVEXChangeControls()` will generate an XVEXControlEvent with state `VEXControlFail`. When the setting is read with `XVEXQueryControls()` or `XVEXGetBlankList()`, described below, a list of zero or more `XVEXBlank` structures will be returned. For each structure, the values are as described below.

`start`
: This element specifies the frame number for the beginning of an unrecorded sequence.

`stop`
: This element specifies the frame number for the end of an unrecorded sequence.

XVEXBlank *XVEXGetBlankList(*display*, *vid*, *nelements*)
    Display **display*;
    VideoID *vid*;
    int **nelements*;

display Specifies the connection to the X server.

vid Specifies the VideoID having the control to be read.

nelements Specifies pointer to the count return value of VEXBlank structures in the returned list.

`XVEXGetBlankList()` returns a pointer to a list of `XVEXBlank` structures. If the value returned in nelements is greater than zero, then memory was allocated and must be freed by the application using XFree().

VEX_DISPLAY_MODE

Some players and recorders can control the way a freeze-frame is displayed. This control allows a client to manipulate the mode.

```
Bool  settable;                    int   mode;
```

The `settable` boolean above is the structure of the control's description data. The `mode` above is the structure of the setting value.

`settable`

If true, then `mode`, below, may be set to either of the described values. If false, then the value of `mode` denotes its permanent value.

`mode`

The value `VEXDisplayField` will display the signal field addressed by the start frame and field with its lines duplicated; e.g., this can provide a frame without temporal information. The value `VEXDisplayFrame` will display two adjacent fields beginning with the field addressed by the start frame and field. Note that the display mode affects the interpretation of the increment for VEX_RECORD.

Bool XVEXDisplayModeSettable(*display*, *vid*)
Display **display*;
VideoID *vid*;

Status XVEXSetDisplayMode(*display*, *vid*, *mode*)
Display **display*;
VideoID *vid*;
int *mode*;

Status XVEXGetDisplayMode(*display*, *vid*, *mode_return*)
Display **display*;
VideoID *vid*;
int **mode_return*;

display Specifies the connection to the X server.

vid Specifies the VideoID having the control to read or write.

mode Specifies the mode to set.

mode_return Returns the current mode.

`XVEXDisplayModeSettable()` returns the `settable` value in the control's description data. `XVEXSetDisplayMode()` sets the frame mode for the device, and `XVEXGetDisplayMode()` returns the current setting for the device's frame mode.

VEX_FIELD_MODE

If a video input or video output device is able to render or capture a single frame, then it is usually able to render or capture a single field, and to control which field. If the device is able, this control will be offered, and it will allow a client to choose whether the device deals in frames or fields. This ability could have been added to the core VEX protocol, but because the feature is not universal, it seemed better to relegate it to controls.

```
int   mode;
```

This control does not have description data. The `mode` above is the structure of the setting value.

`mode`

The value `VEXFieldOne` for a video output will cause a `XVEXCaptureGraphics()` request with full-motion false to capture only field one; the value VEXFieldtwo will capture only field two; the value `VEXFieldBoth` will capture both fields. For a video input, these values will cause a `XVEXRenderVideo()` with full-motion false to render just field one, just field two or both fields, respectively.

Because of the possible scaling and hardware-specific result, precisely which scan lines are rendered to, or which pixels are captured from, becomes difficult to define. This control is intended to define the destination pixels updated by a `XVEXCaptureGraphics()`, and the source pixels that are sampled by a `XVEXRenderVideo()`.

- Setting this control to `VEXFieldOne` will cause a `XVEXRenderVideo()` with full-motion false to use the first field in the video signal as its source. For `XVEXCaptureGraphics()` with full-motion false, only the first field in the destination video signal will be changed.
- Setting this control to `VEXFieldTwo` will cause `XVEXRenderVideo()`, full-motion false, to sample the first field in the video signal, and will cause a `XVEXCaptureGraphics()`, full-motion false, to update the second field in the video signal.
- If this control is set to `VEXFieldBoth`, then both fields will be sampled or updated, as appropriate.

It should be noted that NTSC has a concept of odd and even frames that correspond with the first and second frames described above, and logically, frame one is odd and frame two even. All other standards, such as PAL and SECAM, define the first frame as even and the second as odd. Therefore, the constants described above, `VEXFieldOne` and VexFieldTwo, should be interpreted as the numerical order of the fields for the signal at hand, *not* whether the referenced frame is odd or even.

Status XVEXSetFieldMode(*display*, *vid*, *mode*)
    Display **display*;
    VideoIO *vid*;
    int *mode*;

Status XVEXGetFieldMode(*display*, *vid*, *mode_return*)
    Display **display*;
    VideoIO *vid*;
    int *mode_return*;

display Specifies the connection to the X server.

vid Specifies the VideoIO having the control to read or write.

mode Specifies the mode to set.

mode_return Returns the current mode value.

`XVEXSetFieldMode()` sets the field mode for the device, and
`XVEXGetFieldMode()` returns the current setting for the device's field mode.

VEX_FILL_COLOR

The VEX_FILL_COLOR control allows a client to specify the fill color for undefined regions during a `XVEXRenderVideo()` or `XVEXCaptureGraphics()`. There are two cases that this is designed to address

- A `XVEXRenderVideo()` with a destination whose visual type is VideoGray or VideoColor, where the destination rectangle does not completely fill the destination drawable.
- Any `XVEXCaptureGraphics()` where the destination rectangle does not completely fill the destination screen.

```
typedef struct {
      Bool  fill;
      short red;
      short green;
      short blue;
} XVEXFillColor;
```

There is no description field for this control. The `XVEXFillColor` is the structure of setting's value field.

`fill`
: If this element is true, then the color defined by the elements below is used as the fill color for undefined areas in the destination.

`red`
`green`
`blue`
: These elements define the fill color for undefined areas in the destination.

If a VideoIO has this control, and it is used in with `XVEXRenderVideo()` or `XVEXCaptureGraphics()`, and `fill` is true, then the complement of the destination rectangle plus any other undefined areas in the destination are filled with the specified color. Recall that the destination in a `XVEXRenderVideo()` is a drawable, and the destination in a `XVEXCaptureGraphics()` is a VideoOut; i.e., the video picture.

Status XVEXSetFillColor(*display*, *vid*, *fillp*)
    Display **display*;
    VideoIO   *vid*;
    XVEXFillColor **fillp*;

Status XVEXGetFillColor(*display*, *vid*, *fill_return*)
    Display **display*;
    VideoIO   *vid*;
    XVEXFillColor **fill_return*;

display	Specifies the connection to the X server.
vid	Specifies the VideoIO having the control to read or write.
fillp	Specifies pointer to the `XVEXFillColor` structure to set.
fill_return	Returns the current `XVEXFillColor` value.

`XVEXSetFillColor()` changes the fill color for the named setting.
`XVEXGetFillColor()` obtains the current fill color for the named setting.

VEX_FRAME

If the mode for VEX_ADDRESS_MODE is `VEXFrameAddress`, this control specifies the start and stop points in the device's media, and the current frame where the device's media is resting if it has been stopped, or the frame just played if the media is in motion. The start and stop frame are place-holders for other controls. For example, the VEX_SEGMENT control uses the start and stop frames defined by VEX_FRAME to define a segment.

```
Bool  field_addressable;
```

```
typedef struct {
     long  start_frame,
           stop_frame,
           current_frame;
     char  start_field,
           stop_field,
           current_field;
} XVEXFrameSetting;
```

The `field_addressable` variable is the structure for the control's description data. The `XVEXFrameSetting` is the structure of the data in the setting's value field, and the description for each of its members follows. Note that an `XVEXFrameSetting` is always accompanied by a mask that determines the subset of elements to be set. Note also that the `start`, `stop` and `current` frames are pairs of values; e.g., a start point is specified by `start_frame` and `start_field`.

`field_addressable`

If this is true, then `start_field`, `stop_field` and `current_field` are defined for querying or changing this control. Otherwise, changing their value has no effect and reading their value yields random data.

`start_frame`
`start_field`
`stop_frame`
`stop_field`

These values set the start (inclusive) and stop (exclusive) frame and field for the subsequent play, record and other actions; no transport actions will happen. For the values to be effective, the mask must contain one or both of the bits named `VEXStartPoint` to and `VEXStopPoint` to change the start and stop point, respectively. Frame values may be any positive number. Field values may be `VEXFieldOne` or `VEXFieldTwo`, and correspond to the first and second fields of a frame, sometimes called the odd and even fields, respectively.

`current_frame`
`current_field`

These values control the frame and field displayed or addressed by the device; the mask must contain the bit named `VEXCurrentPoint` to effect a change. If VEX_ADDRESS_MODE is `VEXFrameAddress`, and `VEXCurrentPoint` is asserted, then setting these values will cause the transport, if any, to move to the named position. If VEX_ADDRESS_MODE is `VEXTimecodeAddress` or `VEXNoAddress`, setting these values has no effect. Values are constrained as for start and stop. Setting a value beyond the capacity of the device will generate a XVEXControlEvent with state `VEXControlFail` to be generated; the current frame number should then be reacquired with `XVEXGetFrame()`.

See VEX_FIELD_MODE for a discussion about `VEXFieldOne` and `VEXFieldTwo` with respect to NTSC and other standards.

```
Bool XVEXFieldAddressable(display, vid)
      Display *display;
      VideoID      vid;

Status XVEXSetFrame(display, vid, mask, framep)
      Display *display;
      VideoID      vid;
      unsigned long mask;
      XVEXFrameSetting *framep;

Status XVEXGetFrame(display, vid, frame_return)
      Display *display;
      VideoID      vid;
      XVEXFrameSetting *frame_return;
```

display Specifies the connection to the X server.

vid Specifies the VideoID having the control to read, write, or examine its description.

mask Specifies mask that determines the fields to set.

framep Specifies pointer to the `XVEXFrameSetting` structure to be set.

frame_return Returns the current `XVEXFrameSetting` value.

`XVEXFieldAddressable()` returns the value of the `field_addressable` variable in the control; if the internal library lookup for the id fails, the routine returns false. `XVEXSetFrame()` changes the frame setting for the named VideoID. `XVEXGetFrame()` obtains the frame settings for the named VideoID.

VEX_JOG

The VEX_JOG control allows the client to move the transport relative to the current position as defined by VEX_FRAME or VEX_TIMECODE. If this control is available, jogs can be done forward or backward. If `field_addressable` in the VEX_FRAME control is true, then the relative movements may made in fields, otherwise movements can only be made on whole frames (two fields).

```
typedef struct {
      int   frame_change;
      int   field_change;
} XVEXMovement;
```

There is no description data for this control. The `XVEXMovement` is the structure of the control's setting value.

`frame_change`
: This value specifies the relative movement of the transport; it is added to the current frame. It may be negative or positive.

`field_change`
: If `field_addressable` in the VEX_FRAME or VEX_TIMECODE control is true, then this value is also added to the current field. Only the values 1, 0 and -1 are acceptable.

The current frame is the frame that is being displayed by the device. The jog is accomplished by adding the value of `frame_change` to the frame number and, if `field_addressable` in the VEX_FRAME control is true, adding `field_change` to the field number. If the resulting field is 3, then the field is set to `VEXFieldOne` (1), and the frame number is incremented. If the resulting field is 0, then the field is set to `VEXFieldTwo` (2), and the frame number is decremented.

Status XVEXJog(*display*, *vid*, *movement*)
      Display **display*;
      VideoID *vid*;
      XVEXMovement **movement*;

display Specifies the connection to the X server.

vid Specifies the VideoID having the control to be read.

movement Specifies a pointer to the movement for the transport.

`XVEXJog()` will move the transport to the frame and field addressed by adding `XVEXMovement` as described above.

Note that this setting is write-only; reading the setting will yield random results, hence no library interface is provided.

VEX_OVERLAY

Some players and recorders can display the current frame number or timecode on their output signal. This control allows a client to choose one or the other or neither.

```
typedef struct {                      int   mode;
        Bool  frame_number;
        Bool  time_code;
} XVEXOverlayInfo;
```

The `XVEXOverlayInfo` is the format of the control's description field. Its members describe which features of the OVERLAY control are available for the device. The `mode` variable describes the structure of the control's setting value.

`frame_number`
: If this member is true, then the frame number for the current frame can be overlaid on the device's output signal.

`time_code`
: If this member is true, then the time code for the current frame can be overlaid on the device's output signal.

`mode`
: For the setting, the mode has two unique values. For `VEXFrameDisplay`, the frame number will be displayed. For `VEXTimeDisplay`, the time code will be displayed.

status XVEXGetOverlayInfo(*display*, *control*, *overlay_return*)
    Display **display*;
    XVEXControl **control*;
    XVEXOverlayInfo **overlay_return*;

status XVEXSetOverlay(*display*, *vid*, *mode*)
    Display **display*;
    VideoID *vid*;
    int *mode*;

status XVEXGetOverlay(*display*, *vid*, *mode_return*)
    Display **display*;
    VideoID *vid*;
    int **mode_return*;

display Specifies the connection to the X server.

control Specifies pointer to the control structure where the description is stored.

overlay_return
Returns the overlay information in the control description.

vid Specifies the VideoID having the control to read or write.

mode Specifies the mode to set.

*mode_return*Returns the current mode.

`XVEXGetOverlayInfo()` returns the `XVEXOverlayInfo` stored in the control. `XVEXSetOverlay()` sets the mode for the device, and `XVEXGetOverlay()` returns the current setting for the device's frame mode.

VEX_PLAY

The VEX_PLAY control allows the client to play predefined portions of the device's media. If VEX_SEGMENT is defined for the device, a list of segments numbers may be provided, and optionally looped.

```
typedef struct {
     int max_segments;
     Bool can_loop;
} XVEXSegmentAbility;
```

```
typedef struct {
     int*segments;
     intnsegments;
     Boolloop;
} XVEXPlayList;
```

The `XVEXSegmentAbility` is the structure of the control's description data. The `XVEXPlayList` is the structure of the setting's value.

`max_segments`
: This value describes the maximum number of segments that may be played; it does *not* limit the number of segments that may be defined. The value will not be less than 1.

`can_loop`
: If this value is true, then the `loop` value in `XVEXPlayList` may be optionally set to true. If `can_loop` is false, a setting with `loop` true will play without looping, and a XVEXControlEvent with status `VEXControlFail` will *not* be generated.

`segments`
: This is a pointer to the list of segment numbers to be played.

`nsegments`
: If the this value is 0, it is interpreted as an unqualified "play" command, and the device simply begins to move the transport. In this case, the transport stops at the end of the media or when a VEX_STOP is issued.

`loop`
: If the list is non-zero-length, then the named segments are played in the order listed. And if the boolean `loop` is true, then the list will be repeated indefinitely. The length of the list must be less than or equal to `max_segments`.

Status XVEXGetSegmentAbility(*display*, *vid*, *ability_return*)
    Display **display*;
    VideoID *vid*;
    XVEXSegmentAbility *ability_return;

Status XVEXPlaySegments(*display*, *vid*, *plist*)
    Display **display*;
    VideoID *vid*;
    XVEXPlayList **plist*;

Status XVEXPlay(*display*, *vid*)
    Display **display*;
    VideoID *vid*;

display Specifies the connection to the X server.

vid Specifies the VideoID having the control to read, write, or examine its description.

ability_return
Returns the segment information in the control description.

plist Specifies a pointer to a `XVEXPlayList` structure.

`XVEXGetSegmentAbility()` returns the `XVEXSegmentAbility()` stored in the control. `XVEXPlaySegments()` will play the specified list of segments. `XVEXPlay()` is a shorthand form that simply puts the transport in motion.

This setting is write-only; reading the setting will yield random results, hence no library interface is provided.

VEX_PRODUCT

This control is a pseudo control whose only use is to provide product information that cannot be discerned otherwise. For example, a video input can be labeled using VEX_PRODUCT such that a client can discover, and perhaps display meaningfully, the vendor, the product and so forth. The setting has no value data, so that reading it returns essentially nothing, and writing it is ignored. Only the description data has any value.

This control is required for every video input, output and device.

```
typedef struct {
      char  *class;
      char  *vendor;
      char  *product_name;
      char  *abbreviation;
      char  *position;
} XVEXProduct;
```

The `XVEXProduct` is the structure for the control's description data. Actually, the data is an ISO-Latin-1 string divided into a set of colon-separated fields, and the library interface splits it up into individual strings. The general format of the string is as follows:

```
class:vendor:product_name:abbreviation:position
```

The actual contents of each field is determined by the Vendor and should be described in their own documentation.

`class`

This is one of a set of predefined names, each representing a class of products. It must be selected from among the following list and is required (this list is incomplete).

VIN	video input
VOUT	video output
VDP	video disc player (no recording ability)
VDR	video disc recorder (able to playback and record)
VTR	video tape recorder or video cassette recorder (able to playback and record)
OTHER	not described by the classes above

`vendor`

The maker or re-seller of the product. The field is required.

`product_name`

This should be the complete marketing title for the product, such as *Digital Video Interface, option 9*. The field is required.

`abbreviation`

This should be an abbreviation used in sales literature, such as DVI. The field is optional, i.e., it may be zero length.

`position`

This should be a relative position number such that if a user could use it to distinguish among identical (or nearly identical) devices. For example, if the workstation has three video input boards, they might be labeled 1, 2 and 3 (in three separate VEX_PRODUCT controls). This field must either be empty or consist *only* of decimal digits.

XVEXProduct *XVEXGetProductInfo(*display*, *vid*)
  Display **display*;
  VideoID *vid*;

display Specifies the connection to the X server.

vid Specifies the VideoID having the control whose description is to be examined.

`XVEXGetProductInfo()` returns a pointer to the parsed set of strings. Storage for the structure is allocated by the library and must be freed by the application using XFree().

VEX_RECORD

The VEX_RECORD control allows a client to begin recording, or to record a specific sequence length. This control uses VEX_FRAME or VEX_TIMECODE to define the start and stop points. If this control is not listed for the device, then it is not capable of recording.

```
int     mode;
```

There is no description field for this control. The `mode` element describes the structure of setting's value field.

`mode`

This value specifies how the start and stop points are changed after the recording is complete. If its value is `VEXIncrement`, then when the record operation is complete, the start and stop points in the device's VEX_FRAME or VEX_TIMECODE (depending on VEX_ADDRESS_MODE) will be incremented by the length of the record. For example, if the start point (frame/field) is 1:1 and the stop point is 2:1, then the length if the record is one frame, and the new start and stop points will be 2:1 and 3:1, respectively. If the value is `VEXStatic`, then the start and stop frames will remain unchanged after the record. If the value is `VEXRelative`, then recording is begun at the current frame (instead of the start frame) and continues indefinitely.

The VEX_RECORD control cannot be read: it is write-only. Any query will yield random results.

Status XVEXRecord(*display*, *vid*, *mode*)
Display **display*;
VideoID *vid*;
int *mode*;

display Specifies the connection to the X server.

vid Specifies the VideoID having the control to be written.

mode Specifies the mode for the record, described above.

`XVEXRecord()` will start recording in the specified mode.

VEX_SEGMENT

The VEX_SEGMENT control allows a client to define a set of segments for use with the VEX_PLAY control. Segments, numbered from one upward, may be defined or deleted from any position.

The setting actually consists of several parts: a start and stop point, defined by the VEX_FRAME or VEX_TIMECODE control, depending on the mode; the speed, defined by the VEX_SPEED control; and the segment number, defined by this control.

```
typedef struct {
      int  action;
      int  number;
} XVEXSegment;
```

There is no description field for this control. The `XVEXSegment` is the structure of setting's value field.

`action`

This element specifies the action for the named segment. The segment may defined (`VEXDefineSegment`) or deleted (`VEXDeleteSegment`).

`segment_number`

If this value is greater than zero, it names the segment defined or deleted. If the value is -1 and action is `VEXDeleteSegment`, then all segment definitions for the device are deleted.

The values provided by other controls are used at the time this control is written with `XVEXSetSegment()`.

From the client's perspective, segments are always defined, and have a default value of zero for all elements; i.e., if VEX_PLAY references an undefined segment, it will be as if all values of the segment are zero. A segment, if defined by a client to some non-zero values, will consume server storage until it is deleted, at which time it takes on the default value again. Therefore, it is suggested that when a client begins to use a device (and probably has ownership), all segments should be deleted before defining new ones.

The VEX_SEGMENT control cannot be read, it is write-only. Any query will yield random results.

Status XVEXSetSegment(*display*, *vid*, *segmentp*)
Display **display*;
VideoID *vid*;
XVEXFrameSetting **segmentp*;

display Specifies the connection to the X server.

vid Specifies the VideoID having the control to be written.

segmentp Specifies pointer to the XVEXSetSetment structure to be set.

`XVEXSetSegment()` changes the segment for the named setting.

VEX_SPEED

The VEX_SPEED control advertises the speeds that the transport can play and sets the speed for future VEX_SEGMENT and VEX_PLAY controls. For example, a video disc player may be able to play full, half and quarter speed forward and backward; if the user chooses quarter speed, the next VEX_SEGMENT will define the segment at quarter speed.

```
typedef struct {                       XVEXFraction    speed;
    XVEXFractionRange *full_frame;
    int               nframe;
    XVEXFractionRange *shuttle;
    int               nshuttle;
} XVEXSpeedAbility;
```

The `XVEXSpeedAbility` is the structure for the control's description field. The `XVEXFraction` is the structure of the setting's value field.

`full_frame`
`shuttle`

These are the lists of `XVEXFractionRange` structures that enumerate all possible playback speeds for the device. Any speed from the full-frame list guarantees a consistent picture. The speeds in the shuttle list are probably faster, but the picture may break up, and at the fastest speed, the picture may disappear. On many less expensive tape machines, the fast-forward or rewind is really the fastest forward or backward shuttle speed.

`nframe`
`nshuttle`

These values describe the length of the full_frame and shuttle list, respectively. The value `nframe` is guaranteed to be greater than zero, but nshuttle may be zero.

`speed`

This value describes the current setting for the VEX_SPEED control. Note that when it is set, the transport is not affected; a VEX_PLAY setting must be issued to effect the new speed.

```
XVEXFractionRange *XVEXFullFrameSpeeds(display, vid, nspeeds)
    Display      *display;
    VideoID      vid;
    int      *nspeeds_return;  /* return value */

XVEXFractionRange *XVEXShuttleSpeeds(display, vid, nspeeds)
    Display      *display;
    VideoID      vid;
    int      *nspeeds_return;  /* return value */

XVEXSpeedAbility *XVEXAllSpeeds(display, vid)
    Display      *display;
    VideoID      vid;

Status XVEXSetSpeed(display, vid, speed)
    Display      *display;
    VideoID      vid;
    XVEXFraction      *speed;

Status XVEXGetSpeed(display, vid, speed_return)
    Display      *display;
    VideoID      vid;
    XVEXFraction      *speed_return;
```

display Specifies the connection to the X server.

vid Specifies the VideoID having the control to read, write, or examine its description.

nspeeds_return
Returns the count of speeds.

speed Specifies a pointer to a single fraction.

speed_return Returns the current `XVEXFraction` speed value.

`XVEXFullFrameSpeeds()` extracts the full_frame speeds from the specified control and returns a pointer to a list of `XVEXFractionRange` structures; the number of structures is returned in *nspeeds*. `XVEXShuttleSpeeds()` extracts and returns the shuttle speeds. `XVEXAllSpeeds()` extracts both the full_frame and shuttle speeds from the control and returns a pointer to a `XVEXSpeedAbility` structure containing both.

All three of the above requests allocate memory for the returned structures. The space should be freed by the programmer using XFree().

`XVEXSetSpeed()` sets the current value for the setting and `XVEXGetSpeed()` returns the current setting.

VEX_STOP

The VEX_STOP control stops transport movement started by VEX_RECORD, VEX_PLAY or VEX_BLANK. There is no description data or setting value for this control, and the setting is write-only so that the result of a `XVEXQueryControls()` for this control is undefined.

```
Status XVEXStop(display, vid)
        Display *display;
        VideoID      vid;
```

display Specifies the connection to the X server.

vid Specifies the VideoID having the control to be written.

VEX_TIMECODE

This control is similar to VEX_FRAME: if the mode for VEX_ADDRESS_MODE is `VEXTimecodeAddress`, it specifies the time code of the start and stop point in the device's media, and the current time code where the device's media is resting if it has been stopped, or the time code of the frame just played if the media is in motion.

```
Bool  field_addressable;    typedef struct {
                                    char  start_hour,
                                          stop_hour,
                                          current_hour;
                                    char  start_minute,
                                          stop_minute,
                                          current_minute;
                                    char  start_second,
                                          stop_second,
                                          current_second;
                                    char  start_frame,
                                          stop_frame,
                                          current_frame;
                                    char  start_field,
                                          stop_field,
                                          current_field;
                            } XVEXTimecodeSetting;
```

The `field_addressable` variable is the structure for the control's description data. The `XVEXTimecodeSetting` is the structure of the data in the setting's value field, and the description for each of its members follows. Note that an `XVEXTimecodeSetting` is always accompanied by a mask that determines the subset of elements to be set or changed. Note also that the `start`, `stop` and `current` times are sets of five values; e.g. a start point is specified by `start_hour`, `start_minute`, `start_second`, `start_frame` and `start_field`.

`field_addressable`

If this is true, then `start_field`, `stop_field` and `current_field` are defined for querying or changing this control. Otherwise, changing their value has no effect and reading their value yields random data.

`start_hour`
`start_minute`
`start_second`
`start_frame`
`start_field`
`stop_hour`
`stop_minute`
`stop_second`
`stop_frame`
`stop_field`

These values set the start (inclusive) and stop (exclusive) times for the subsequent play, record and other actions; no transport actions will happen. For the values to be effective, the mask must contain one or both of the bits named `VEXStartPoint` and `VEXStopPoint` to change the start and stop point, respectively. Time values are constrained as follows: hour 0-23; minute 0-59; second 0-59; frame 0-29 (0-24 for PAL and SECAM); field values may be `VEXFieldOne` or `VEXFieldTwo`, and correspond to the first and second fields of a frame, sometime called the odd and even fields, respectively.

`current_hour`
`current_minute`
`current_second`
`current_frame`
`current_field`

These values control the frame and field displayed or addressed by the device. If VEX_ADDRESS_MODE is `VEXTimecodeAddress`, then setting these values will cause the transport, if any, to move to the named position. If VEX_ADDRESS_MODE is `VEXFrameAddress` or `VEXNoAddress`, setting these values has no effect. Values are constrained as for start and stop. Setting a value beyond the capacity of the device will cause a XVEXControlEvent with state `VEXControlFail` to be generated; the current frame number should then be reacquired with `XVEXGetTimecode()`.

See VEX_FIELD_MODE for a discussion about `VEXFieldOne` and `VEXFieldTwo` with respect to NTSC and other standards.

Bool XVEXFieldAddressable(*display*, *vid*)
    Display **display*;
    VideoID *vid*;

Status XVEXSetTimecode(*display*, *vid*, *mask*, *timep*)
    Display **display*;
    VideoID *vid*;
    unsigned long *mask*;
    XVEXTimecodeSetting **timep*;

Status XVEXGetTimecode(*display*, *vid*, *time_return*)
    Display **display*;
    VideoID *vid*;
    XVEXTimecodeSetting **time_return*;

display Specifies the connection to the X server.

vid Specifies the VideoID having the control to read, write, or examine its description.

mask Specifies mask that determines the fields to set.

timep Specifies pointer to the `XVEXTimecodeSetting` structure to be set.

time_return Returns the current `XVEXTimecodeSetting` value.

`XVEXFieldAddressable()` returns the value of the `field_addressable` variable in the control. Note that `XVEXFieldAddressable()` is used for both VEX_FRAME and VEX_TIMECODE because if it is true for one it is true for the other. If only one of the controls is available for the VideoID, the library interface uses the correct one. `XVEXSetTimecode()` changes the time code setting for the named setting. `XVEXGetTimecode()` obtains the time code settings for the named setting (which contains the device).

VEX_VIDEO_ADJUSTMENT

This control specifies a variety of adjustments for the decoding or digitization of a video signal for a video input. Some devices will be able to adjust attributes such as contrast and hue while others may not be able to adjust hue (NTSC defines hue, but PAL does not). If any of the controls described below are available for a video input, then this control will be present for the video input.

```
typedef struct {
    XVEXFractionRange *saturation;
    int               n_sat;
    XVEXFractionRange *contrast;
    int               n_cont;
    XVEXFractionRange *luma_gain;
    int               n_luma;
    XVEXFractionRange *chroma_gain;
    int               n_chroma;
    XVEXFractionRange *hue;
    int               n_hue;
    XVEXFractionRange *bright;
    int               n_bright;
} XVEXVideoAbilities;
```

```
typedef struct {
    XVEXFraction saturation;
    XVEXFraction contrast;
    XVEXFraction luma_gain;
    XVEXFraction chroma_gain;
    XVEXFraction hue;
    XVEXFraction bright;
} XVEXVideoAdjustments;
```

The `XVEXVideoAbilities` is the structure for the control's description data. The `XVEXVideoAdjustments` is the structure of the data in the setting's value field. Note that an `XVEXVideoAdjustments` is always accompanied by a mask that determines the subset of elements to be set or changed.

`n_sat`
`n_cont`
`n_luma`
`n_chroma`
`n_hue`
`n_bright`

These values state the length of the corresponding `XVEXFractionRange` structures. If the control is available, at least one of the lists will have a non zero length.

`saturation`

The `XVEXFractionRange` with this name describes the possible values for the saturation setting. The `XVEXFraction` with this name is the structure of the setting value. Saturation is the amount of color information present in the video image. Small values result in black and white images; large values provide increasing amounts of color. For the setting, the mask must contain the bit named `VEXSaturation`.

`contrast`

The `XVEXFractionRange` with this name describes the possible values for the contrast setting. The `XVEXFraction` with this name is the structure of the setting value. Contrast determines the dynamic range of information present in the video image. Small values result in an image of constant color and intensity; large values produce more contrast. For the setting, the mask must contain the bit named `VEXContrast`.

`luma_gain`

The `XVEXFractionRange` with this name describes the possible values for the luma_gain setting. The `XVEXFraction` with this name is the structure of the setting value. Luminance gain determines how much the luminance portion of the video image is amplified. Small values result in ?; large values produce ?. For the setting, the mask must contain the bit named `VEXLuminance`.

`chroma_gain`
: The `XVEXFractionRange` with this name describes the possible values for the chroma_gain setting. The `XVEXFraction` with this name is the structure of the setting value. Chrominance gain determines how much the chrominance portion of the video image is amplified. Small values result in ?; large values produce ?. For the setting, the mask must contain the bit named `VEXChrominance`.

`hue` The `XVEXFractionRange` with this name describes the possible values for the hue setting. The `XVEXFraction` with this name is the structure of the setting value. Hue shifts the phase of the incoming video's color information relative to its reference, causing a shift in color. Small values result in ?; large values produce ?. For the setting, the mask must contain the bit named `VEXHue`.

`bright`
: The `XVEXFractionRange` with this name describes the possible values for the bright setting. The `XVEXFraction` with this name is the structure of the setting value. Brightness determines the black level of the video signal. Small values result in ?; large values produce ?. For the setting, the mask must contain the bit named `VEXBright`.

Status XVEXGetVideoAbilities(*display*, *vid*, *ability_return*)
    Display **display*;
    VideoID   *vid*;
    XVEXVideoAbilities**ability_return*;

Status XVEXGetVideoAdjustment(*display*, *vid*, *adjust_return*)
    Display **display*;
    VideoID   *vid*;
    XVEXVideoAdjustments **adjust_return*;

Status XVEXSetVideoAdjustment(*display*, *vid*, *mask*, *adjust*)
    Display **display*;
    VideoID   *vid*;
    unsigned long *mask*;
    XVEXVideoAdjustments **adjust*;

display Specifies the connection to the X server.

control Specifies pointer to the control structure.

ability_return
: Returns the video adjustment info from the control description.

vid Specifies the VideoID having the control to read, write, or examine its description.

mask Specifies mask that determines the fields to set.

adjust_return
: Returns the current `XVEXVideoAdjustments` value.

adjust Specifies pointer to the `XVEXVideoAdjustments` structure to be filled in or set.

`XVEXGetVideoAbilities()` returns the value of the `XVEXVideoAbilities` structure in the control. `XVEXGetVideoAdjustment()` obtains the video adjustment for the named setting (which contains the device). `XVEXSetVideoAdjustment()` changes the video adjustment for the named setting.

4. Encoding

The entries here follow the encoding specified by the VEX Protocol Encoding document. In the table below, the order of the elements and the *size* are precisely the same as the protocol encoding. The types and names for the control and setting elements are taken from the protocol specification; the names for the sub-elements of the description field and setting data match the C structure elements from this document.

VEX_ADDRESS_MODE Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_ADDRESS_MODE"
4	VIDEODEVICE	id	assigned
1	n/a	type	2 (Device)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	32 (LISTofINT32)
1	unused		
4	CARD32	setting-length	4
4	CARD32	description-length	0

VEX_ADDRESS_MODE Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_ADDRESS_MODE"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	4
4	CARD32	mode	VEXFrameAddress (1) VEXTimecodeAddress (2) VEXNoAddress (3)

VEX_BLANK_SEARCH Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_BLANK_SEARCH"
4	VIDEODEVICE	id	assigned
1	n/a	type	2 (Device)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	32 (LISTofINT32)
1	unused		
4	CARD32	setting-length	0
4	CARD32	description-length	0

VEX_BLANK_SEARCH Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_BLANK_SEARCH"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	8*n (n is the number of blank sequences returned)
4	CARD32	start	
4	CARD32	stop	

VEX_DISPLAY_MODE Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_DISPLAY_MODE"
4	VIDEODEVICE	id	assigned
1	n/a	type	2 (Device)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	32 (LISTofINT32)
1	unused		
4	CARD32	setting-length	4
4	CARD32	description-length	0

VEX_DISPLAY_MODE Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_DISPLAY_MODE"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	4
4	CARD32	mode	VEXDisplayFrame (1) VEXDisplayField (2)

VEX_FIELD_MODE Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_FIELD_MODE"
4	VIDEODEVICE	id	assigned
1	n/a	type	0 (Input) or 1 (Output)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	32 (LISTofINT32)
1	unused		
4	CARD32	setting-length	4
4	CARD32	description-length	0

VEX_FIELD_MODE Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_FIELD_MODE"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	4
4	CARD32	mode	VEXFieldOne (1) VEXFieldTwo (2) VEXFieldBoth (2)

VEX_FILL_COLOR Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_FILL_COLOR"
4	VIDEODEVICE	id	assigned
1	n/a	type	0 (Input) or 1 (Output)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	32 (LISTofINT32)
1	unused		
4	CARD32	setting-length	8
4	CARD32	description-length	0

VEX_FILL_COLOR Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_FILL_COLOR"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	8
2	CARD16	fill	True or False
2	CARD16	red	
2	CARD16	green	
2	CARD16	blue	

VEX_FRAME Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_FRAME"
4	VIDEODEVICE	id	assigned
1	n/a	type	2 (Device)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	8 (LISTofINT8)
1	unused		
4	CARD32	setting-length	0
4	CARD32	description-length	1
1	BOOL	field_addressable	true or false
3	pad		

VEX_FRAME Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_FRAME"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	4+8*n (n is number of elements selected by the mask)
4	CARD32	mask	0x01 (VEXStartPoint) 0x02 (VEXStopPoint) 0x04 (VEXCurrentPoint)
4	CARD32	start_frame	
4	CARD32	start_field	1 (VEXFieldOne) or 2 (VEXFieldTwo)
4	CARD32	stop_frame	
4	CARD32	stop_field	1 (VEXFieldOne) or 2 (VEXFieldTwo)
4	CARD32	current_frame	
4	CARD32	current_field	1 (VEXFieldOne) or 2 (VEXFieldTwo)

VEX_JOG Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_JOG"
4	VIDEODEVICE	id	assigned
1	n/a	type	2 (Device)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	32 (LISTofINT32)
1	unused		
4	CARD32	setting-length	8
4	CARD32	description-length	0

VEX_JOG Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_JOG"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	8
4	CARD32	frame_change	
4	CARD32	field_change	1, 0 or -1

VEX_OVERLAY Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_OVERLAY"
4	VIDEODEVICE	id	assigned
1	n/a	type	2 (Device)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	32 (LISTofINT32)
1	unused		
4	CARD32	setting-length	4
4	CARD32	description-length	0
1	BOOL	frame_number	True or False
1	BOOL	time_code	True or False

VEX_OVERLAY Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_OVERLAY"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	4
4	CARD32	mode	VEXFrameDisplay 1 VEXTimeDisplay 2

VEX_PLAY Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_PLAY"
4	VIDEODEVICE	id	assigned
1	n/a	type	2 (Device)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	32 (LISTofINT32)
1	unused		
4	CARD32	setting-length	0
4	CARD32	description-length	8
4	CARD32	max-segments	
4	CARD32	can-loop	

VEX_PLAY Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_PLAY"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	4+4*n (n is number of segments)
4	CARD32	loop	True or False
4*n	CARD32	segment	

VEX_PRODUCT Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_PRODUCT"
4	VIDEODEVICE	id	assigned
1	n/a	type	0 (Input) or 1 (Output) or 2 (Device)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	8 (LISTofINT8)
1	unused		
4	CARD32	setting-length	0
4	CARD32	description-length	n (length of null-terminated string)
n	CARD8	string	

VEX_RECORD Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_RECORD"
4	VIDEODEVICE	id	assigned
1	n/a	type	2 (Device)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	32 (LISTofINT32)
1	unused		
4	CARD32	setting-length	4
4	CARD32	description-length	0

VEX_RECORD Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_RECORD"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	4
4	CARD32	mode	VEXIncrement (1) VEXStatic (2) VEXRelative (3)

VEX_SEGMENT Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_SEGMENT"
4	VIDEODEVICE	id	assigned
1	n/a	type	2 (Device)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	32 (LISTofINT32)
1	unused		
4	CARD32	setting-length	8
4	CARD32	description-length	0

VEX_SEGMENT Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_SEGMENT"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	8
4	CARD32	action	1 (VEXDefineSegment) or 2 (VEXDeleteSegment)
4	CARD32	segment_number	-1 (delete all segments) or n (n > 0)

VEX_SPEED Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_SPEED"
4	VIDEODEVICE	id	assigned
1	n/a	type	2 (Device)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	32 (LISTofINT32)
1	unused		
4	CARD32	setting-length	8
4	CARD32	description-length	8 + 32*m + 32*n (m is number of full-frame fraction ranges, n is number of shuttle fraction ranges)
4	CARD32	n-full-frames	
4	CARD32	n-shuttles	
4	INT32	num-base	
4	INT32	num-inc	
4	INT32	num-limit	
4	INT32	denom-base	
4	INT32	denom-inc	
4	INT32	denom-limit	
4	n/a	num_type	0 (Geometric) or 1 (Linear)
4	n/a	denom_type	0 (Geometric) or 1 (Linear)

VEX_SPEED Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_SPEED"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	8
4	CARD32	numerator	
4	CARD32	denominator	

VEX_STOP Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_STOP"
4	VIDEODEVICE	id	assigned
1	n/a	type	2 (Device)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	32 (LISTofINT32)
1	unused		
4	CARD32	setting-length	0
4	CARD32	description-length	0

VEX_STOP Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_STOP"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	0

VEX_TIMECODE Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_TIMECODE"
4	VIDEODEVICE	id	assigned
1	n/a	type	2 (Device)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	8 (LISTofINT8)
1	unused		
4	CARD32	setting-length	0
4	CARD32	description-length	1
1	BOOL	field_addressable	true or false
3	pad		

VEX_TIMECODE Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_TIMECODE"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	1+5*n (n is number of elements selected by the mask)
4	CARD8	mask	0x01 (VEXStartPoint) 0x02 (VEXStopPoint) 0x04 (VEXCurrentPoint)
4	CARD8	start_hour	0 through 23
4	CARD8	start_minute	0 through 59
4	CARD8	start_second	0 through 59
4	CARD8	start_frame	0 through 29
4	CARD8	start_field	1 (VEXFieldOne) or 2 (VEXFieldTwo)
4	CARD8	stop_hour	0 through 23
4	CARD8	stop_minute	0 through 59
4	CARD8	stop_second	0 through 59
4	CARD8	stop_frame	0 through 29
4	CARD8	stop_field	1 (VEXFieldOne) or 2 (VEXFieldTwo)
4	CARD8	current_hour	0 through 23
4	CARD8	current_minute	0 through 59
4	CARD8	current_second	0 through 59
4	CARD8	current_frame	0 through 29
4	CARD8	current_field	1 (VEXFieldOne) or 2 (VEXFieldTwo)

VEX_VIDEO_ADJUSTMENT Control

size	*type*	*name*	*value*
4	ATOM	type	"VEX_VIDEO_ADJUSTMENT"
4	VIDEODEVICE	id	assigned
1	n/a	type	0 (Input)
1	n/a	setting-format	32 (LISTofINT32)
1	n/a	description-format	32 (LISTofINT32)
1	unused		
4	CARD32	setting-length	0
4	CARD32	description-length	24 + 32*(i+j+k+l+m+n) (i through k are the number of saturation, contrast, luma_gain, chroma_gain, hue and bright fraction ranges)
4	CARD32	n_sat	
4	CARD32	n_cont	
4	CARD32	n_luma	
4	CARD32	n_chroma	
4	CARD32	n_hue	
4	CARD32	n_bright	
4	INT32	num-base	
4	INT32	num-inc	
4	INT32	num-limit	
4	INT32	denom-base	
4	INT32	denom-inc	
4	INT32	denom-limit	
4	n/a	num_type	0 (Geometric) or 1 (Linear)
4	n/a	denom_type	0 (Geometric) or 1 (Linear)

VEX_VIDEO_ADJUSTMENT Setting

size	*type*	*name*	*value*
4	ATOM	type	"VEX_VIDEO_ADJUSTMENT"
4	VIDEODEVICE	id	assigned
1	n/a	format	32 (LISTofINT32)
3	unused		
4	CARD32	length	4+8*n (n is number of one bits in mask)
4	CARD32	mask	(0x01) VEXSaturation (0x02) VEXContrast (0x04) VEXLuminance (0x08) VEXChrominance (0x10) VEXHue (0x20) VEXBright
4	CARD32	sat numerator	
4	CARD32	sat denominator	
4	CARD32	cont numerator	
4	CARD32	cont denominator	
4	CARD32	luma numerator	
4	CARD32	luma denominator	
4	CARD32	chroma numerator	
4	CARD32	chroma denominator	
4	CARD32	hue numerator	
4	CARD32	hue denominator	
4	CARD32	bright numerator	
4	CARD32	bright denominator	