V++ Video Driver Writer's Guide

Digital Optics

PO Box 35-715 • Browns Bay • Auckland 1330 • New Zealand
Phone +64 (21) 736 378, +64 (9) 478 5779 • Fax +64 (9) 479 4750
www.digitaloptics.co.nz

V++ Video Driver Writer's Guide

Second Edition (Online v2.3)
Produced in New Zealand.

This is the online edition of the *V++ Video Driver Writer's Guide* and is formatted for US Letter paper.

V++ is subject to continuous improvement therefore Digital Optics reserves the right to modify its specifications at any time and without notice. This manual is intended to be a fair representation of certain features and capabilities of V++ but discrepancies may occur from time to time as development progresses. Nothing in this document shall be construed as a commitment by Digital Optics to implement or support any particular feature.

Table of Contents

1. Introduction

V++ incorporates an industry leading video architecture which provides for the easy integration of almost any kind of video capture hardware, including frame grabbers and digital cameras. This guide is the official programmer's documentation for the V++ video architecture and describes how you can write drivers to integrate your own equipment into the system.

You can choose your own level of complexity when you write a video driver. There are 8 mandatory functions that you must include and you can choose to implement more, up to a total of 47, to provide better support for your hardware.

When V++ loads your driver it automatically determines what level of support it provides and makes the best use it can of whatever functions are available.

Some of the important features of the video architecture for developers are:

- Built-in GUI support for all of the most common video capture operations
- VPascal language support for all video operations
- Hot loading and unloading of drivers (no restart required)
- Simultaneous access to multiple devices
- Ability to call a custom configuration dialog for each device
- Ability to add a custom control interface to your driver
- Ability to send custom commands direct to your driver
- Most driver functions are optional – only 8 are required for a minimal driver

Your driver is automatically integrated into the V++ interface and incorporated in features like the VideoBar™ and the multi-device tabbed dialog boxes.

Before You Start

This guide assumes that you are broadly familiar with V++ and with general concepts of digital imaging. For information about how to use video devices as an end-user, see the V++ online help. You will need to be familiar with the Video dialog boxes in order to test the driver. To write a V++ video driver, you will be required to write a standard Windows DLL with a programming language of your choice.

Terminology

The following are definitions of important terms used throughout this guide:

Driver
A video driver is a Windows DLL, written according to this specification, which exports a set of functions to support one or more individual video devices. It is not a device driver in the Windows sense of the term and devices will usually still need their proprietary drivers to be installed. Video drivers often simply act as an intermediate layer between V++ and a hardware API library.

Device
A video device is a single piece of equipment that captures video images – it is most likely to be a video frame grabber or a digital camera. It may use any of a number of methods to transfer images to the driver, including ISA bus, PCI bus, serial, parallel, USB, Firewire and others. As far as V++ is concerned, the hardware technology is completely encapsulated by the driver.

Hot Loading
V++ loads and unloads video drivers at runtime without having to shutdown and restart. When a driver loads or unloads, all user interface elements (the VideoBar™, tabbed dialogs etc) automatically update to accommodate the changes. This flexible process is referred to as "hot loading".

Support

If you have questions about implementing a V++ video driver that are not addressed by this document then please contact the Digital Optics technical support department at the following address:

video@digitaloptics.co.nz

For general information about V++ and downloads (including free video drivers) please refer to our web site:

http://www.digitaloptics.co.nz

2. Driver Overview

Concepts

V++ supports a variety of function categories for video operations. Each of these requires a certain set of functions to be present in the video driver and will only be available if the driver supports the necessary subsets of functions.

Basic Operations

As a minimum, a video driver must implement functions to perform the following tasks:

- Initialization and shut down
- Return the name of the device it controls
- Provide status and error codes
- Get the dimensions and bit depth of the video frame
- Perform a frame capture operation
- Copy a captured video frame into host memory

These tasks are performed by the 8 mandatory functions that your driver is obliged to export.

Additional Operations

The driver may also implement additional functions to perform various other tasks, depending on the capabilities of the devices it controls. The following types of operations are supported by the V++ video driver model:

- Control and selection of multiple devices
- Displaying a custom configuration dialog box
- Asynchronous video capture
- Frame buffer direct memory access
- Region of interest capture
- Continuous frame capture
- Controlling multiple video input channels
- Control of integrating cameras
- Externally triggered frame capture
- Hardware supported sequence acquisition
- Providing a custom operational interface to a device
- Returning custom driver error messages
- Returning a driver name, provider and version number
- Returning hardware and driver performance information
- Executing commands sent directly to the driver

Any particular operation will automatically appear in V++ provided sufficient supporting functions have been exported by the video driver.

Supporting Multiple Devices

The video architecture inherently supports multi-device and multi-driver operation. This means that V++ can load multiple video drivers simultaneously (each of which supports a different device) and also that any one driver may support more than one device.

If your driver supports multiple devices then it must export some additional functions that enable V++ to determine what devices are available and to select them individually. Once a device is selected it remains so until another device is selected. Devices are referred to by a zero-based index number.

The driver must apply all video functions to its currently selected device.

Programming Languages

Video drivers can be written in any programming language that supports Windows 32-bit DLLs and the standard calling model. The reference chapters in this guide provide sample declarations in both Pascal and C.

Calling Conventions

All exported functions must be compiled with standard calling conventions – in many languages this is referred to as "stdcall". V++ imports functions by name so you must export the function names exactly as shown in this documentation and preserve case. DO NOT use C++ name mangling. Refer to your compiler's documentation if you are unsure about how to set the calling model.

Exporting Functions

V++ determines what the capabilities of your driver are by examining which functions the DLL has exported. To be recognized as a V++ video driver it must at least export the 8 mandatory functions described in chapter 4. All other functions are optional and the ones you choose to export will determine how much support is available for the video device in V++.

Data Types

The following data types are used for parameters or function results:

Pascal	*C/C++*	*Description*
integer	int	32-bit signed integer
cardinal	uint	32-bit unsigned integer
pointer	void *	32-bit pointer
PChar	char *	32-bit pointer to a character string
boolean	int	32-bit true/false value
HWnd	hwnd	Windows handle
TRect	TRect	Windows rectangle structure (with "+1" conventions[1]) Consists of four 32-bit integers: Left, Top, Right, and Bottom.
TVideoEvent	void *	32-bit pointer to a V++ callback function

Hot Loading

To ensure that your driver is fully compatible with hot loading, do not include automatic initialization and finalization code in the DLL. All initialization and clean-up should be done in the `OpenDriver` and `CloseDriver` functions that every driver must implement.

[1] The rectangle "+1" conventions used by Windows means that the TRect is defined so that: Right = Left + Width and Bottom = Top + Height

Memory Layout

Functions that return image or sequence data do so in a raw packed format with no headers and no padding. V++ allocates memory for functions that need it and the driver must write any image data to the addresses provided. Image data is stored row-wise as packed bytes, for example:

- A monochrome image of 640 × 480 with 8-bits per pixel is stored as 480 rows of 640 bytes, making a total of exactly 307,200 bytes. The first 640 bytes contain the first row and so on.

- A color image is stored in a similar way except that each pixel consists of 3 components – one each for red, green and blue. For a 640 × 480 image with 24-bits per pixel, the data would be stored as 480 rows of 1,920 bytes (3 × 640), for a total of 921,600 bytes. Each pixel consists of three bytes stored in the order R, G, B – do not use the DIB reversed component order.

Do not write past the end of memory buffers supplied by V++ – this would almost certainly result in an access violation error in the driver.

Registry Settings

There are numerous settings that a video driver might need to store in the registry, depending on the function it exports. V++ automatically stores ROI information for each device in the registry but all other settings that need to be preserved between sessions are the responsibility of the driver.

We recommend that you store driver settings in your own area of the registry, using the per user settings branch. For example, if a company called "XYZ Inc." produces a driver for a device called "VideoMaster" then its root key in the registry would be similar to the following:

```
HKEY_CURRENT_USER\Software\XYZ Inc\VideoMaster
```

You should store settings and configuration information when `CloseDriver` is called or whenever something changes. These settings should be restored on startup, when `OpenDriver` is called.

Loading and Installing a Driver

Installing a video driver in V++ is very simple and can be done manually, by the end-user, or automatically with an installation program.

Manual Installation

To install a video driver manually, copy all of its files to a folder on the hard disk and use the Load button on the Video | System dialog box to locate and load the primary DLL file. This is described in more detail in the V++ online help (search for "video" then "Getting Started with Video"). Once the driver has been installed it will be loaded at startup on subsequent sessions. Drivers can be unloaded from the Video | System dialog box.

Automatic Installation

You can use an install program to install your driver's files on the user's hard disk and to make registry settings that tell V++ to load it at startup. You can also make a registry setting to ensure that the V++ video acquisition add-in is enabled.

The base registry location, or "base key", for V++ settings is as follows:

```
HKEY_CURRENT_USER\Software\Digital Optics\V++
```

All V++ settings are stored at this location in a sub-key named according to the version number. For example, for V++ 4.0 the registry settings are stored under the following key:

```
HKEY_CURRENT_USER\Software\Digital Optics\V++\4.0
```

The version string for the latest V++ installation is stored as a REG_SZ string called "Version" under the base key. This allows you to automatically make driver settings in the correct registry sub-tree no matter what version is installed. If the "Version" value is missing then use the default version string "4.0".

The list of drivers to be loaded is stored in the following registry location (for version 4.0):

HKEY_CURRENT_USER\Software\Digital Optics\V++\4.0\Add-Ins\Video\Drivers

Under that key you will find a string entry for each driver, containing the full path of the main DLL file. The names of the string entries are "Driver0", "Driver1" and so on.

To add your own driver to the list, first create an appropriate string entry with any name that is not already in use. It does not have to form part of the sequence used by default. For example, an installer for a device called "VideoMaster" might add an entry called "VideoMaster" containing the path information.

When V++ next runs, it loads all of the drivers with an entry under this registry key and converts the list to its preferred format if there are any names not in the standard sequence.

Secondly, you should ensure that the V++ video acquisition add-in is enabled by checking a registry setting under the following key (for version 4.0):

HKEY_CURRENT_USER\Software\Digital Optics\V++\4.0\Add-Ins\Video

The "Enabled" value at this location determines whether or not video support is turned on. Your installer should ensure that the video acquisition system is enabled and that drivers will be loaded by setting Enabled to an integer value of 1.

IMPORTANT NOTES

- Do not make any entries in the "…\Add-Ins\Video\Devices" branch of the registry as V++ uses this to store its own information about available devices.

- The first version of V++ with video capabilities was 4.0.5.68

Modifications

This specification for a V++ video driver may change from time to time, usually because new functions have been added. All such new functions will be optional so that old drivers remain compatible with future releases of V++. This also means that older versions of V++ will always be able to load newer drivers.

This is the second edition of the specification. The following new functions were added:

```
CaptureSequenceEx
GetMaxFrameRate
GetMinTimeout
IsOsSupported
```

See later chapters for full information.

3. Function Categories

This section summarizes all of the function groups that a driver may implement. Only the first category is mandatory although some of the others are highly recommended (see Recommendations at the end of this chapter). Detailed function definitions are provided in the chapters that follow.

Basic Functions

The following 8 functions are mandatory and must be implemented by every V++ video driver:

`OpenDriver`	Initialize the driver and any subordinate libraries it may use
`CloseDriver`	Clean up internal structures and shutdown the driver
`GetDeviceName`	Return the name of the selected video device
`DeviceReady`	Return true or false to indicate whether the device is ready for use
`DeviceError`	Return a code indicating the error condition
`GetFrameInfo`	Return the x-size, y-size, bit depth and sample count of the video frame
`Snapshot`	Capture and store a single video frame
`ReadFrame`	Copy the stored video frame into host memory (provided by V++)

System Functions

If the driver supports multiple devices then it must implement all of the following functions:

`GetDeviceCount`	Return the total number of devices supported by the driver and available
`SelectDevice`	Select a device by its zero-based index number
`SelectedDevice`	Return the index number of the currently selected device

Driver Information

The following optional functions provide information about the driver. If available, the information is displayed by the Video | System dialog box:

`GetDriverName`	Return the driver's name (may be different to the device name)
`GetDriverProvider`	Return the name of the driver developer
`GetDriverVersion`	Return a version number string for the driver

Implementing these functions is recommended as it is very easy to do and provides valuable feedback to the V++ user.

Performance Information

The driver may implement the following functions to provide information to V++ about the performance characteristics of the hardware and the driver:

`GetMaxFrameRate`	Return the maximum supported frame rate
`GetMinTimeout`	Return the minimum recommended capture timeout
`IsOsSupported`	Indicates whether the driver works under the installed operating system

These functions are supported by V++ 4.0.5.94 and later releases.

Region of Interest

If there is hardware support for region of interest frame capture then the driver may implement the following routines:

`SetROI`	Set a rectangular region of interest within the video frame
`GetROI`	Return the current region of interest
`ReadROI`	Copy image data from the region of interest into a memory buffer

If the functions above are not implemented, then V++ will simulate ROI support if the `ReadBuffer` function is available (see below).

Asynchronous Capture

To enable V++ to perform asynchronous video capture operations the driver must implement the following functions:

`ASyncCapture`	Start a frame capture operation and return immediately
`ASyncStop`	Perform any clean up tasks required following an asynchronous capture
`ASyncStatus`	Return a code indicating the asynchronous capture status
`ASyncAbort`	Terminate any pending capture operations

The minimum subset of these routines required for asynchronous operations is `ASyncCapture` and `ASyncStatus` – implementing these functions is highly recommended for optimum performance.

Frame Buffer

The following functions, if implemented, return extended information about the video frame and provide direct access to frame buffer memory on the device.

`GetSampleFormat`	Return a code indicating the interpretation of pixel samples
`GetPixelAspect`	Return the physical aspect ratio of the pixels
`ReadBuffer`	Copy image data from a region in the frame buffer to a memory buffer
`WriteBuffer`	Copy image data from host memory into a region in the frame buffer

These functions do not have to be implemented as a group so you may implement only those that you wish to provide.

Input Channels

Many frame grabbers support more than one video input channel. If these are software selectable then the driver may implement the following control functions:

`GetChannelCount`	Return the number of video channels available on the device
`SelectChannel`	Switch to a channel by its zero-based index number
`SelectedChannel`	Return the index number of the currently selected input channel

Continuous Capture

The following routines may be implemented to support continuous capture, or "video streaming", on devices that are capable of it. Streaming involves continuously capturing frames and transferring them to an internal or hardware buffer.

`SetContinuous`	Turn continuous capture mode on or off
`GetContinuous`	Return the current status of continuous capture mode

Integration

Certain frame grabbers and cameras support on-chip integration (ie. selectable exposure times). To support integrating cameras the driver must implement the following functions:

`SetExposureTime`	Set the exposure time for an integrating camera
`GetExposureTime`	Get the current exposure time setting

Exposure Modes

If alternate exposure modes are available then the driver must implement the following routines to support them. At present, the only options are normal exposure or externally triggered exposure.

`SetExposureMode`	Set the exposure mode to normal or triggered
`GetExposureMode`	Get the current exposure mode setting
`SetTriggerTimeout`	Set the timeout for a triggered exposure
`GetTriggerTimeout`	Get the current trigger timeout value

Sequences

If the device supports hardware sequence acquisition then the driver may implement this using the following function:

`CaptureSequence`	Perform a hardware assisted sequence capture into pre-allocated memory
`CaptureSequenceEx`	Perform a hardware assisted sequence capture into pre-allocated memory

The `CaptureSequenceEx` function supercedes the `CaptureSequence` function although both are still recognised. See the Sequences chapter for more information.

Note that these routines should only be implemented to provide hardware assistance for sequence capture. If neither is implemented then V++ provides timing for the capture of sequences. If the driver supports asynchronous capture and the device is fast then V++ can capture video sequences in real time, even without hardware assistance.

Digital I/O Lines

Many frame grabbers and digital cameras incorporate TTL input and/or output lines for integration with other apparatus. To make this capability available in V++ the driver must implement one or both of the following functions:

`ReadTTL`	Read the states of the TTL inputs
`WriteTTL`	Write to the TTL outputs

User Interface Support

A video driver may implement up to two custom dialog boxes – one for hardware configuration and one for control of specialized hardware, as follows. It is highly recommended that you implement `ShowConfigForm` in your video drivers. However, most drivers will not need to implement `ShowCustomForm`.

`ShowConfigForm`	Display a modal dialog box for hardware configuration
`ShowCustomForm`	Display a modal or modeless dialog box for hardware control

Drivers may define their own error messages to go with the codes they return in the `ErrorCode` function. To do so, implement the following function

`DeviceMessage`	Convert an error code (from DeviceError) into an error message

V++ provides default messages if this function is not implemented. However, if `ErrorCode` returns non-standard codes then it is recommended that the driver implements `DeviceMessage` as well.

Command Interface

Drivers may implement a command interface for any purpose they may require (eg. debugging, support for specialized features, quick configuration etc…) using the following function:

`Execute`	Execute a custom command sent direct to the driver

This is an excellent way to extend VPascal support for specialized hardware.

Recommendations

In addition to the 8 mandatory functions, it is highly recommended that you implement the following optional functions:

```
GetDriverName
GetDriverProvider
GetDriverVersion
GetMinTimeout
ASyncCapture
ASyncStatus
ReadBuffer
ShowConfigForm
```

Although not compulsory, these optional functions enable V++ to provide better performance with any video device.

4. Basic Functions

Basic functions are mandatory and must all be implemented by every V++ video driver. They represent the minimum functionality required to integrate into the V++ video system.

OpenDriver

Initialize the hardware, the driver and any subordinate libraries

Declaration

```
procedure OpenDriver ;
void OpenDriver() ;
```

Details

This function is called immediately after V++ loads the driver. Put all initializations here rather than in a DLL initialization section.

CloseDriver

Clean up internal structures and shutdown the driver

Declaration

```
procedure CloseDriver ;
void CloseDriver() ;
```

Details

This function is called when V++ terminates or unloads the driver. Use this function for cleaning up and freeing resources used by the DLL. Avoid using a DLL exit procedure.

GetDeviceName

Return the name of the selected video device

Declaration

```
function GetDeviceName( Name:PChar; nChars:integer ) : PChar ;
char *GetDeviceName( char *Name; int nChars ) ;
```

Parameters

`Name`	Pointer to a string buffer into which the device name is to be written
`nChars`	The maximum string length the buffer can take

Return Value

For convenience, the function should return the pointer `Name`.

Details

The name returned by this function is the device name used on tabbed video dialogs and on the individual bands of the VideoBar™.

DeviceReady

Return true or false to indicate whether the device is ready for use

Declaration

```
function DeviceReady : boolean ;
int DeviceReady() ;
```

Details

Return true (1) if the selected video device is ready to digitize a frame. If the device can't be detected or is busy then return false (0).

DeviceError

Return a code indicating the error condition

Declaration

```
function DeviceError : integer ;
int DeviceError() ;
```

Details

Return an error code for the selected device or 0 if there is no error condition. If you implement `DeviceMessage` then you can choose your own error codes, otherwise use the following:

0	There are no errors
1	Timeout error waiting for an operation
2	No response from device

You must always return 0 to indicate that there is no error – even if you have chosen your own error codes.

GetFrameInfo

Return the x-size, y-size, bit depth and sample count of the video frame

Declaration

```
procedure GetFrameInfo( var xSize,ySize,BitDepth,Samples:integer ) ;
void GetFrameInfo( int *xSize, int *ySize, int *BitDepth, int *Samples ) ;
```

Parameters

`xSize`	Set to the x-size of the video frame
`ySize`	Set to the y-size of the video frame
`BitDepth`	Set to the bit depth of each sample
`Samples`	Set to the number of samples per pixel

Details

Monochrome devices should return with Samples = 1 and RGB color devices should return with Samples = 3. For example, a 24-bit RGB color frame grabber returns BitDepth = 8 (as each component is 8 bits) and Samples = 3.

If the sample format is other than unsigned integer then you should implement `GetSampleFormat`. This enables you to implement drivers that capture signed integer and floating point data.

Snapshot

Capture and store a single video frame

Declaration

```
procedure Snapshot ;
void Snapshot() ;
```

Details

This is the most basic function of a video capture device – to digitize and store a single frame. This function should wait until the digitization is complete before returning and the image should be stored in the device frame buffer. If your device cannot store the image then it is up to the driver to buffer the data until V++ reads it out.

ReadFrame

Copy the stored video frame into host memory (provided by V++)

Declaration

```
procedure ReadFrame( Data:pointer ) ;
void ReadFrame( void *Data ) ;
```

Parameters

`Data` Pointer to an image buffer created by V++

Details

The entire video frame buffer should be copied into the host buffer pointed to by `Data`. The host buffer will be large enough to take one full frame of video data. See chapter 2 for information about image memory layout.

5. System Functions

System functions provide for supporting more than one device with a single video driver. V++ can use all devices supported by a driver and can load more than one driver simultaneously. If your driver supports multiple video devices then you must implement all of the following functions.

GetDeviceCount

Return the total number of devices supported by the driver

Declaration

```
function GetDeviceCount : integer ;
int GetDeviceCount() ;
```

Details

This function must return the number of devices that are currently available through this driver. Do not include devices that are potentially available – only those that can be accessed right now. This total should be fixed once `OpenDriver` has been called.

If N devices are available then they must be numbered from 0 to N-1. The driver must keep track of the currently selected device and ensure that all other functions operate exclusively on the selected device.

SelectDevice

Select a device by its zero-based index number

Declaration

```
procedure SelectDevice( Index:integer ) ;
void SelectDevice( int Index ) ;
```

Parameter

Index The zero-based index of the device

Details

If there is an attempt to select a non-existent device it should simply be ignored.

SelectedDevice

Return the index number of the currently selected device

Declaration

```
function SelectedDevice : integer ;
int SelectedDevice() ;
```

6. Driver Information

The optional functions described here provide information about the driver that is displayed by the Video | System dialog box. Implementing these functions is recommended as it is very easy to do and provides valuable feedback to the V++ user.

GetDriverName

Return the driver's name (may be different to the device name)

Declaration

```
function GetDriverName( Name:PChar; nChars:integer ) : PChar ;
char *GetDriverName( char *Name; int nChars ) ;
```

Parameters

`Name`	Pointer to a string buffer into which the driver name is to be written
`nChars`	The maximum string length the buffer can take

Return Value

For convenience, the function should return the pointer `Name`.

Details

The driver may have a name that differs from the device it controls (particularly if it controls multiple devices). The name returned by this function is used to identify the driver in the list of loaded drivers on the Video | System dialog box.

GetDriverProvider

Return the name of the driver developer

Declaration

```
function GetDriverProvider( Name:PChar; nChars:integer ) : PChar ;
char *GetDriverProvider( char *Name; int nChars ) ;
```

Parameters

`Name`	Pointer to a string buffer into which the provider name is to be written
`nChars`	The maximum string length the buffer can take

Return Value

For convenience, the function should return the pointer `Name`.

Details

The result of this function is displayed in the "Provider" column of the Video | System driver list. Return your own name of that of your company.

GetDriverVersion

Return a version number string for the driver

Declaration

```
function GetDriverVersion( Version:PChar; nChars:integer ) : PChar ;
char *GetDriverVersion( char *Version; int nChars ) ;
```

Parameters

`Version`	Pointer to a string buffer into which the device version is to be written
`nChars`	The maximum string length the buffer can take

Return Value

For convenience, the function should return the pointer `Version`.

Details

Create and return a formatted string indicating the version number of your driver. It will be displayed in the appropriate column of the Video | System driver list.

7. Performance Information

The driver may implement several functions to provide information to V++ about the performance characteristics of the hardware and the driver.

The performance information functions are supported by V++ 4.0.5.94 and later releases. Earlier releases will simply ignore these functions.

GetMaxFrameRate

Return the maximum frame rate supported by the hardware and driver

Declaration

```
function GetMaxFrameRate : integer ;
int GetMaxFrameRate() ;
```

Details

This function should return the maximum possible frame rate that the selected device could be expected to achieve. The return value is not expected to be a guarantee of performance but is used to set parameter limits in video dialog boxes.

GetMinTimeout

Return the minimum recommended capture timeout for the selected device

Declaration

```
function GetMinTimeout : integer ;
int GetMinTimeout() ;
```

Details

Where possible, V++ performs capture operations using the asynchronous functions in order to impose an overall timeout in the case of unresponsive hardware. This function should be implemented if a frame capture may take longer than a typical video frame time. Returning a more suitable timeout value than the default eliminates spurious timeout errors.

This function should take into account the exposure mode – if the triggered exposure mode is in effect then the trigger timeout should be returned if it is greater than the normal timeout.

IsOsSupported

Return true or false to indicate whether the driver can operate in the installed operating system

Declaration

```
function IsOsSupported : boolean ;
int IsOsSupported() ;
```

Return Value

Returns true (1) if the installed operating system is supported and false (0) otherwise

Details

Some drivers may be restricted to run only in specific operating systems. In these cases, this function should be implemented so that V++ can determine the OS compatibility at load time. The driver should use the Windows `GetVersionEx()` API function to find out what operating system is running and return a true or false result accordingly.

8. Region of Interest

V++ supports region of interest (ROI) processing for video capture if there is sufficient driver support. The functions described here are mainly intended for devices that have hardware support for region of interest frame capture.

V++ automatically provides ROI functionality if either:

a) the ROI functions described here are implemented, or
b) the `ReadBuffer` function is implemented

The simplest way to enable ROI support for your video device is simply to implement `ReadBuffer`.

If an ROI is defined and enabled for a particular device then all subsequent capture operations on that device will produce images from the ROI.

V++ preserves ROI settings, on a per device basis, between sessions. Therefore, your driver does not need to keep track of these settings.

SetROI

Set a rectangular region of interest within the video frame

Declaration

```
procedure SetROI( Rect:TRect ) ;
void SetROI( TRect Rect ) ;
```

Parameter

`Rect` The rectangle definition (+1) describing the required ROI

Details

This function is used to set the region of interest boundaries for the device. Note that the rectangle is defined as Windows TRect, and therefore:

Rect.Right = Rect.Left + Width
Rect.Bottom = Rect.Top + Height

where Width and Height are the inclusive dimensions of the ROI.

IMPORTANT
If an empty rectangle is passed in (ie. one where Left = Right and Top = Bottom) then this cancels the ROI and indicates that the driver should revert to full frame capture.

GetROI

Return the current region of interest within the video frame

Declaration

```
function GetROI : TRect ;
TRect GetROI() ;
```

Return Value

The current region of interest (see SetROI for more information). If no ROI is selected then this function should return an empty rectangle.

ReadROI

Copy image data from the region of interest into a memory buffer

Declaration

```
procedure ReadROI( Data:pointer ) ;
void ReadROI( void *Data ) ;
```

Parameters

`Data` Pointer to an image buffer created by V++

Details

Only the image data from the current region of interest should be copied into the buffer pointed to by `Data`. If no ROI is defined then this function should do nothing. See chapter 2 for information about image memory layout.

9. Asynchronous Capture

Asynchronous video capture operations are those that return immediately – without waiting for the frame capture to complete. This enables the calling application to perform other tasks while a frame capture is in progress and to readout the data when it is ready.

To enable V++ to perform asynchronous video capture operations the driver must implement the functions described here. Only `ASyncCapture` and `ASyncStatus` are required to enable asynchronous operation – implementing these two functions is highly recommended for optimum performance.

ASyncCapture

Start a frame capture operation and return immediately

Declaration

```
procedure ASyncCapture ;
void ASyncCapture() ;
```

Details

This function performs a similar task to `Snapshot` but does not wait for the frame capture to complete. Capturing one frame of NTSC video can take between 33 and 66 ms depending on whether or not the device needs to wait for the start of the next frame. Implementing this function (along with `ASyncStatus`) enables V++ to perform other tasks during this time.

ASyncStatus

Return a code indicating the asynchronous capture status

Declaration

```
function ASyncStatus : integer ;
int ASyncStatus() ;
```

Return Value

An integer status code chosen from the following:

0 Ready to readout
1 Capture pending
2 Capture in progress

Details

When the device is idle, whether or not `ASyncCapture` has been called, this function should return 0 to indicate that ReadFrame or a similar function may be called. If the device is busy then return either 1 or 2 to indicate that data is about to be acquired. It is not strictly necessary to distinguish between the two cases as some frame grabbers do not differentiate between the pending and in-progress states.

ASyncAbort

Terminate any pending capture operations

Declaration

```
procedure ASyncAbort ;
void ASyncAbort() ;
```

Details

Implementing this function is optional but may be useful if supported by hardware. If this function were called, the intent would be to abandon the capture operation and not to read out any data.

ASyncStop

Perform any clean up tasks required following an asynchronous capture

Declaration

```
procedure ASyncStop ;
void ASyncStop() ;
```

Details

Some frame grabbers require capture operations to be explicitly finished – this function can be implemented if that is the case for your device. However, most drivers will not need to implement this function.

10. Frame Buffer Functions

The following functions, if implemented, return extended information about the video frame and provide direct access to frame buffer memory on the device. These functions do not have to be implemented as a group so you may implement only those that you wish to provide.

It is highly recommended that you implement the `ReadBuffer` function for any device capable of direct access to the frame buffer memory.

GetPixelAspect

Return the physical aspect ratio of the pixels

Declaration

```
procedure GetPixelAspect( var xa,ya:integer ) ;
void GetPixelAspect( int *xa, int *ya ) ;
```

Parameters

xa Horizontal pixel dimension
ya Vertical pixel dimension

Details

Some frame grabbers digitize a standard 4:3 video signal into a square memory block. This results in unequal sampling rates in the horizontal and vertical directions, ie. non-square pixels. This function must return the physical aspect ratio of the pixels relative to each other. For example, if the device digitizes a video image into a 512 x 512 memory buffer then the aspect ratio is 4:3 and the function should return xa = 4 and ya = 3.

GetSampleFormat

Return a code indicating the interpretation of pixel samples

Declaration

```
function GetSampleFormat : integer ;
int GetSampleFormat() ;
```

Return Value

The sample format code, as follows:

0 Default V++ format (unsigned integer)
1 Unsigned integer
2 Signed integer
3 Floating-point (IEEE)

Details

The majority of video capture devices return unsigned integer data at a variety of bit depths. For devices that return other data formats, implement this function and return the appropriate code. The format codes refer to an individual sample – in the case of an RGB color image, a sample means one of the three components that make up each pixel.

ReadBuffer

Copy image data from a region in the frame buffer to a memory buffer

Declaration

```
procedure ReadBuffer( Rect:TRect; Data:pointer ) ;
void ReadBuffer( TRect Rect, void *Data ) ;
```

Parameters

`Rect` The rectangular region to read data from
`Data` Pointer to an image buffer created by V++

Details

Almost all video capture devices have a frame buffer memory. This function must copy image data directly from the frame buffer into a host memory buffer. The region is defined as a Windows TRect and therefore:

Rect.Right = Rect.Left + Width
Rect.Bottom = Rect.Top + Height

where Width and Height are the inclusive dimensions of the region. If an empty rectangle is passed in (ie. one where Left = Right and Top = Bottom) then no data should be copied.

Only data from the specified region of the frame buffer can be copied. Refer to chapter 2 for more information about memory layout.

IMPORTANT
This function is used to simulate ROI functionality if no ROI specific routines are available. It is therefore highly recommended that drivers implement this function.

WriteBuffer

Copy image data from host memory into a region in the frame buffer

Declaration

```
procedure WriteBuffer( Rect:TRect; Data:pointer ) ;
void WriteBuffer( TRect Rect, void *Data ) ;
```

Parameters

`Rect` The rectangular region to write data to
`Data` Pointer to an image buffer created by V++

Details

Almost all video capture devices have a frame buffer memory. This function is used to write image data directly into the frame buffer. Its main importance is for frame grabbers that provide a video output that can be connected directly to either an auxiliary monitor or a video printer.

Most drivers do not need to implement this function.

11. Input Channels

Many frame grabbers support more than one video input channel. If these are software selectable then the driver may implement the control functions described here.

Input channels must be numbered from 0.

GetChannelCount

Return the number of video channels available on the device

Declaration

```
function GetChannelCount : integer ;
int GetChannelCount() ;
```

Return Value

The number of video input channels available.

SelectChannel

Switch to a channel by its zero-based index number

Declaration

```
procedure SelectChannel( Index:integer ) ;
void SelectChannel( int Index ) ;
```

Parameter

The zero-based channel index number

Details

The driver is responsible for maintaining channel selection between sessions. However, it is acceptable to always default to channel 0 on startup.

SelectedChannel

Return the index number of the currently selected input channel

Declaration

```
function SelectedChannel : integer ;
int SelectedChannel() ;
```

Return Value

The channel index number

12. Continuous Capture

Continuous capture is a mode in which the video device continuously captures frames and transfers them to a memory buffer. In modern cameras, this kind of operation may be called "video streaming" but in older frame grabbers it was often called "continuous grab".

Continuous capture mode is not used for sequence capture, even at real time rates, and therefore most drivers will not need to implement it. However, there are some situations where it is likely to be very useful, for example:

- When working with a camera that needs to enter video streaming mode in order to support asynchronous capture (some industrial Firewire cameras are in this category).
- With an older frame grabber in order to support real time focussing on an auxiliary monitor

IMPORTANT
The driver must ensure that all other functions continue to operate as expected if continuous grab mode is enabled. For example, if continuous capture is turned on then the asynchronous capture routines (see chapter 9) may need to operate differently.

SetContinuous

Start or stop continuous capture mode

Declaration

```
procedure SetContinuous( State:boolean ) ;
void SetContinuous( int State ) ;
```

Parameter

`State` Set true (1) to turn on continuous capture and false (0) to turn it off

GetContinuous

Return the current state of continuous capture mode

Declaration

```
function GetContinuous : boolean ;
int GetContinuous() ;
```

Return Value

The present state of continuous capture mode. Return true (1) if continuous capture is turned on and return false (0) otherwise.

13. Integration

Certain frame grabbers and cameras support on-chip integration (ie. selectable exposure times). Typically, a special video camera is required with a control line that determines the length of the exposure. The integration control line is operated either by the host computer or by a compatible frame grabber.

To support integrating cameras the driver must implement the following functions:

SetExposureTime

Set the exposure time for an integrating camera

Declaration

```
procedure SetExposureTime( msTime:integer ) ;
void SetExposureTime( int msTime ) ;
```

Parameter

`msTime` The exposure time in milliseconds

Details

Setting the exposure time to 0 must cancel the integration mode and revert to standard video exposure times.

GetExposureTime

Get the current exposure time setting

Declaration

```
function GetExposureTime : integer ;
int GetExposureTime() ;
```

Return Value

The current exposure time in milliseconds

14. Exposure Modes

If alternate exposure modes are available then the driver must implement the following routines to support them. At present, the only options are normal exposure or externally triggered exposure.

SetExposureMode

Set the exposure mode to normal or triggered

Declaration

```
procedure SetExposureMode( Mode:integer ) ;
void SetExposureMode( int Mode ) ;
```

Parameter

`Mode` The required exposure mode, as follows:

0	Normal exposure
1	Externally triggered exposure

Details

The only exposure modes available at present are normal and triggered, however additional modes may be implemented in future. If external triggering is supported then it is advisable to implement the `SetTriggerTimeout` function as well.

GetExposureMode

Get the current exposure mode setting

Declaration

```
function GetExposureMode : integer ;
int GetExposureMode() ;
```

Return Value

The current exposure mode, as defined in `SetExposureMode`.

SetTriggerTimeout

Set the timeout for a triggered exposure

Declaration

```
procedure SetTriggerTimeout( msTime:integer ) ;
void SetTriggerTimeout( int msTime ) ;
```

Parameter

`msTime` The timeout delay in milliseconds

Details

If a device is waiting for an external trigger before performing a capture operation then it is possible for the driver to hang if the trigger never comes. To provide for a timeout delay on triggered exposures then implement this function. Provided `SetTriggerTimeout` and `GetTriggerTimeout` are

available then V++ will monitor the timeout for you wherever possible. In some situations (such as a snapshot) the driver must monitor the timeout itself.

GetTriggerTimeout

Get the current trigger timeout value

Declaration

```
function GetTriggerTimeout : integer ;
int GetTriggerTimeout() ;
```

Return Value

The timeout delay in milliseconds

15. Sequences

If a video device provides hardware assistance for fast sequence capture then you can make this available to V++ by implementing the `CaptureSequence` and `CaptureSequenceEx` functions.

> NOTE
> The `CaptureSequence` routine has been superceded by `CaptureSequenceEx` which returns the actual frame rate achieved (it is otherwise identical to `CaptureSequence`). The newer `CaptureSequenceEx` function is recognised by V++ 4.0.5.94 and later releases but drivers should implement both routines if they wish to remain compatible with earlier versions.

CaptureSequence

Perform a hardware assisted sequence capture into pre-allocated memory
This routine has been superceded by `CaptureSequenceEx` (see below)

Declaration

```
procedure CaptureSequence( nFrames,msPeriod:integer; Data:pointer ) ;
void CaptureSequence( int nFrames, int msPeriod, void *Data ) ;
```

Parameters

`nFrames`	The total number of frames to capture
`msPeriod`	The time between frames, in milliseconds
`Data`	Pointer to a sequence buffer created by V++

Details

Implement this function if the video device provides hardware assistance for fast sequence capture. If `msPeriod` is zero (or simply less than the minimum possible frame time) then perform a maximum rate capture.

If an ROI is defined then each sequence frame must confirm to the ROI boundaries, otherwise each frame must be full size. The buffer pointed to by `Data` will be large enough to store exactly `nFrames` frames. See chapter 2 for information about memory layout.

V++ provides real time sequence timing if this function is not implemented (provided the hardware is fast enough).

CaptureSequenceEx

Perform a hardware assisted sequence capture into pre-allocated memory
This routine replaces the older `CaptureSequence` routine (defined above)

Declaration

```
function CaptureSequenceEx( nFrames,msPeriod:integer; Data:pointer ) : integer ;
int CaptureSequenceEx( int nFrames, int msPeriod, void *Data ) ;
```

Parameters

`nFrames`	The total number of frames to capture
`msPeriod`	The time between frames, in milliseconds
`Data`	Pointer to a sequence buffer created by V++

Return Value
The actual frame rate achieved during the sequence acquisition (in frames/second)

Details
Implement this function if the video device provides hardware assistance for fast sequence capture. If `msPeriod` is zero (or simply less than the minimum possible frame time) then perform a maximum rate capture.

If an ROI is defined then each sequence frame must confirm to the ROI boundaries, otherwise each frame must be full size. The buffer pointed to by `Data` will be large enough to store exactly `nFrames` frames. See chapter 2 for information about memory layout.

This function should compute and return the actual frame rate achieved when the sequence was captured. If you do not wish to implement this computation then either return zero or the inverse of the frame delay time passed in (`msPeriod`).

V++ provides real time sequence timing if this function is not implemented (provided the hardware is fast enough).

16. Digital I/O Lines

Many frame grabbers and digital cameras incorporate TTL input and/or output lines for integration with other apparatus. To make this capability available in V++ the driver must implement one or both of the following functions.

You may support up to 32 input and/or output TTL lines which are mapped to individual bits in the values read and written by these functions.

ReadTTL

Read the states of the TTL inputs

Declaration

```
function ReadTTL : integer ;
int ReadTTL() ;
```

Return Value

An integer (32-bits) in which each bit indicates the value of one of the TTL input lines, numbered from 0 to 31.

Details

If there are fewer than 32 lines then simply set the bits corresponding to the lines you do have.

WriteTTL

Write to the TTL outputs

Declaration

```
procedure WriteTTL( Value:integer ) ;
void WriteTTL( int Value ) ;
```

Parameter

`Value` An integer (32-bits) in which each bit indicates the value to be written to one of the TTL input lines, numbered from 0 to 31.

Details

If there are fewer than 32 lines then simply set the bits corresponding to the lines you do have.

17. User Interface Support

A video driver may implement up to two custom dialog boxes – one for hardware configuration and one for general control of specialized hardware, by implementing the following functions. It is highly recommended that you implement `ShowConfigForm` in your video drivers. However, most drivers will not need to implement `ShowCustomForm`.

In addition, drivers may implement the `DeviceMessage` function to define their own error messages to go with the codes they return in the `ErrorCode` function.

DeviceMessage

Convert an error code (from `DeviceError`) into an error message

Declaration

```
function DeviceMessage( Msg:PChar; nChars,Code:integer ) : PChar ;
char *DeviceMessage( char *Msg, int nChars, int Code ) ;
```

Parameters

`Msg`	Pointer to a string buffer into which the error message is to be written
`nChars`	The maximum string length the buffer can take
`Code`	The error code

Return Value

For convenience, the function should return the pointer `Msg`.

Details

V++ provides default messages if this function is not implemented. However, if `ErrorCode` returns non-standard codes then it is recommended that the driver implements `DeviceMessage` as well.

ShowConfigForm

Display a modal dialog box for hardware configuration

Declaration

```
function ShowConfigForm( Parent:HWnd ) : integer ;
int ShowConfigForm( hwnd Parent ) ;
```

Parameter

`Parent` The Windows handle of the V++ parent window

Return Value

The modal result of the dialog box

Details

It is highly recommended that you implement this function so that the user can configure the video device inside the V++ environment. Drivers should write any configuration changes to the registry.

The dialog box can include any kind of capability you may require but should limit its actions to configuring the hardware. The dialog may change any of the visible frame capture settings, including the dimensions of the video frame itself, as V++ will automatically update when the function returns.

The configuration must be modal, ie. you must not be able to return to V++ without closing the dialog.

If you are writing your driver in Borland Delphi or Borland C++ Builder then use the following outline to write this function:

- Set the driver's `Application.Handle` property to the value of `Parent`
- Create the form with the local `Application` object as the owner
- Set the `FormStyle` to `fsStayOnTop`
- Display the form using `ShowModal`
- Free the form
- Return the form's `ModalResult` as the function result

In other languages, use the `Parent` parameter as the parent window of the dialog box.

IMPORTANT
Be sure to always return the correct modal result (ie. `id_Ok` if something has changed, `id_Cancel` otherwise) so that V++ settings remain synchronized with driver settings.

ShowCustomForm

Display a modal or modeless dialog box for customized hardware control

Declaration

```
function ShowCustomForm( Parent:HWnd; Id:integer; Notify:TVideoEvent ) : boolean ;
int ShowCustomForm( hwnd Parent, int Id, TVideoEvent Notify ) ;
```

Parameters

`Parent`	The Windows handle of the V++ parent window
`Id`	Driver identifier
`Notify`	Pointer to a V++ callback function (details below)

Return Value

Return true (1) if V++ should readout a video frame on return and false (0) otherwise.

Details

This function enables you to implement a complete custom user interface for your video driver. Most drivers have no need to implement `ShowCustomForm` but you may wish to do so if you have specialized requirements or wish to extend the V++ video user interface.

If this function is implemented then V++ displays a "Control" command on the Video menu that enables you to launch it. Similarly, an additional button appears on the appropriate section of the VideoBar™.

The dialog box displayed by `ShowCustomForm` may be either modal or modeless and can perform any kind of video operation at all, including capturing video data and making changes to driver settings (for example, changing video input channels or the ROI definition). Although it is preferable to implement hardware configuration using the `ShowConfigForm` function, it is acceptable to do so as part of this function also.

If the form is modeless then `ShowCustomForm` can return immediately with a false (0) result. The dialog will stay on screen and can trigger readouts into V++ using the `Notify` callback function described below.

If you are writing your driver in Borland Delphi or Borland C++ Builder then use the following outline to write this function:

- Set the driver's `Application.Handle` property to the value of `Parent`
- Create the form with the local `Application` object as the owner
- Set the `FormStyle` to `fsStayOnTop` (optional)
- Display the form using `Show` or `ShowModal`
- Return from `ShowCustomForm`
- Free the form either when it is closed or when `CloseDriver` is called

In other languages, use the `Parent` parameter as the parent window of the dialog box.

IMPORTANT – V++ Callback

The `Notify` parameter is a pointer to a callback function in V++ that the dialog box can use to request readout or to indicate that device settings have been changed. It enables the driver's custom interface to perform repeated capture operations and to keep V++ synchronized with changes it makes to settings.

The callback function is defined as follows:

```
procedure VideoCallback( Id,Event:integer ) ;
void VideoCallback( int Id, int Event ) ;
```

It is compiled with the same "standard call" conventions used in the drivers themselves.

When calling the callback function, pass in the `Id` value that was passed as a parameter to `ShowCustomForm`. This is a unique identifier that enables V++ to determine which specific driver is requesting service. The `Event` parameter indicates what kind of service is required. The following events are defined:

0 Data is available in the frame buffer for readout

This event causes V++ to readout and display the image stored in the frame buffer of the driver's currently selected video device. If an ROI is defined and enabled then only that area will be read.

1 Device settings have been changed

Use this event to notify V++ that device settings have changed. This ensures that settings displayed in V++ will remain synchronized with the driver. This is not necessary for custom settings of which V++ is unaware.

2 Call a shared VPascal procedure

If a VPascal procedure with the DDE share name "Video" exists then this event causes it to be executed. For more complex interactions the driver should use a full DDE implementation rather than a callback.

18. Command Interface

Drivers may implement a command interface for any purpose they may require (eg. debugging, support for specialized features, quick configuration etc…) using the function described below.

Execute

Execute a custom command sent direct to the driver

Declaration

```
function Execute( Cmd:PChar; Param:cardinal ) : integer ;
int Execute( PChar Cmd, uint Param ) ;
```

Parameters

`Cmd` Pointer to a string containing a driver command
`Param` 32-bit unsigned integer

Return Value

32-bit signed integer

Details

The `Execute` command provides a way for commands to be sent directly to the driver and may be used for any purpose the driver writer requires. The interpretation of the `Cmd` and `Param` parameters is entirely up to the driver. The return value may or may not be meaningful depending on the driver.

An important application for `Execute` is to extend the range of video functions available in VPascal. For example, if your driver implements a custom interface for a specialized video device (using `ShowCustomForm`) then there may be features that you also wish to expose in VPascal. By implementing `Execute` you can make some or all of your custom features available to VPascal programmers (through the VPascal `vidDriverCommand` function).

It is also possible for a user to issue commands to the driver manually using the Commands button on the Video | System dialog box. This launches a dialog box that allows you to send commands to the driver's `Execute` function, if it has one. This can be very valuable for debugging during the development of a driver.

Appendix A: Focus Issues

Timing for the V++ video focus dialog box is generated in a special thread which instructs the dialog box to capture and display frames and synchronizes the display with other GUI activity. The timing thread does not use a Windows multimedia timer (because these cause problems if latency is high) and may appear to use a lot of processor time, however it does make significant provisions for other processes to run.

Although there is no actual "focus loop", it is instructive to consider the sequence of events as similar to a loop structured as follows:

```
SetContinuous( true )
ASyncCapture()
Repeat
  Check ASyncStatus() until it returns vid_Ready
  ASyncStop()
  Read image data
  ASyncCapture()
  Update the display
  Check for errors
Until focus is halted
ASyncAbort()
SetContinuous( false )
```

Notes

- V++ attempts to use asynchronous capture for focussing, in order to increase performance. In particular, it updates the display with frame N while frame N+1 is being captured. If you don't implement asynchronous capture then focus will be significantly slower.

- For devices that are capable of continuous capture, or streaming, you can improve focus performance by making this mode available in your driver. If implemented, continuous capture mode is turned on before focussing starts and turned off when it is halted.

- Focus can be halted either by the user or automatically due to errors reported by the driver. By default, a certain number of errors will be tolerated before focus is halted and the tolerance level can be modified in the V++ registry (see Appendix B).

- If focus is halted while a capture operation is still pending then ASyncAbort() will be called. You can force V++ to call ASyncAbort() every time focus is halted by making a registry setting (see Appendix B).

- It is acceptable for the dimensions of the image, and even its data type, to change from frame to frame while focussing.

Appendix B: V++ Registry Settings

There are a number of registry settings that control the behavior of V++ in video operations. Many of these are simply the stored values of parameters that the user can set using V++ dialog boxes. However, others are intended for expert use only and may be useful to driver writers. The undocumented registry settings are described below.

Value Name	*Type*	*Description*
ActualRate	REG_DWORD	Set this value true (1) to force V++ to set the frame rate of a captured sequence to the actual rate achieved rather than to the requested frame rate. Default is false (0).
AlwaysAbort	REG_DWORD	If this value is true (1) then V++ will always call ASyncAbort() when a focus operation halts. By default, the abort call only takes place if a frame capture is pending when focus is halted.
DriverSequence	REG_DWORD	This true/false value controls whether or not V++ calls the driver's sequence capture functions. If it is false (0) then sequence capture timing is done internally. The default setting is true (1).
FocusPeriod	REG_DWORD	Limits the focus frame rate by setting the minimum inter-frame time, in milliseconds. Default is 33 ms.
FocusTolerance	REG_DWORD	Indicates the number of driver errors that will be tolerated while focus is taking place. Set to 0 if you want to abort focus on the first error. Default is 5.
GrabTimeout	REG_DWORD	Sets a timeout value, in milliseconds, after which frame capture operations will be aborted. Note that V++ also takes into account the exposure time, trigger timeout setting and the return value of GetMinTimeout(), if implemented, when computing the actual timeout to use. Default is 100 ms.
MaxFrameRate	REG_DWORD	The maximum frame rate that can be selected in V++ dialog boxes. If the driver implements GetMaxFrameRate() then its return value overrides this setting. Default is 30 frames per second.

The settings described above are all found in the following registry key (for V++ 4.0):

HKEY_CURRENT_USER\Software\Digital Optics\V++\4.0\Add-Ins\Video

Note that the key path includes the version number of the installed V++ software.

Index