Indigo Accessories SDK 1.4

Programmer’s Guide

Table of Contents

Part I – Introduction

The Indigo Accessories SDK provides a standardized interface across a collection of PC hardware communications platforms. In addition, it provides support for all current types of Indigo IR cameras by way of a "camera type" enumeration. The usefulness of the SDK comes from its abstraction of applicable hardware details so as to eliminate the need for hardware-specific programming at the application level.

As a result of using this SDK, applications get the advantage of both a convenient programming model, as well as a platform-independent method of switching between distinct communication hardware platforms (i.e. "accessories") whenever necessary.

The API published by the SDK supports the following four distinct interfaces:

a. **Discovery** – provides for dynamic discovery of available interfaces and devices.
b. **Command & Control** – enables camera control & status via serial command sequences.
c. **Video** – enables real-time digital video acquisition.
d. **Configuration** – enables hardware-specific configuration of communication devices.

Depending on its capabilities, it is possible for a given hardware platform to implement (usually in the form of a DLL) all of the above four interfaces, or a chosen subset from the set (b, c, d). Interface (a) is mandatory for every hardware-specific module.

Please see Appendix A for a list of currently supported hardware platforms.

SDK Architecture

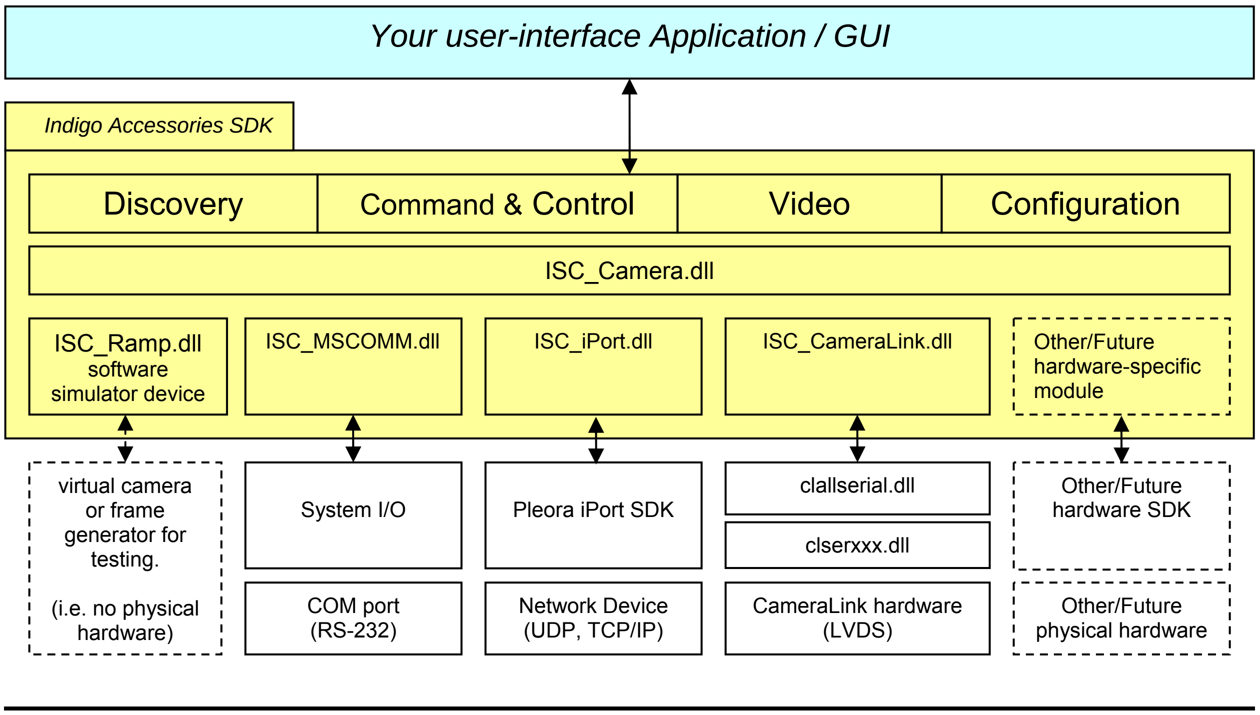
Your user-interface Application / GUI
Indigo Accessories SDK
Discovery
Command & Control
Video
Configuration
ISC_Camera.dll
ISC_Ramp.dll software simulator device
ISC_MSCOMM.dll
ISC_iPort.dll
ISC_CameraLink.dll
Other/Future hardware-specific module
virtual camera or frame generator for testing. (i.e. no physical hardware)
System I/O
Pleora iPort SDK
clallserial.dll
clserxxx.dll
Other/Future hardware SDK
COM port (RS-232)
Network Device (UDP, TCP/IP)
CameraLink hardware (LVDS)
Other/Future physical hardware

Figure 1

As illustrated in Figure 1, the SDK consists of several software abstraction layers and hardware-specific modules. This architecture provides for the creation of a single standardized API for a client application called the ISC_Camera interface. The ISC_Camera module implements the top-level interface of the SDK that interfaces to applications.

Currently, the SDK supports certain contemporary hardware interfaces, such as AIA-standard Camera Link, "iPort" network-based frame grabber, Firewire, etc. It also provides some software simulator devices. Additionally, the SDK is extensible to future hardware platforms by simply adding the necessary hardware-specific modules.

Each hardware-specific module of the SDK has a set of *mandatory* functions, and a set of *optional* functions. Applications can check the availability of optional functions before trying to invoke them. Please refer to the header file ISC_Camera.h for all API function declarations.

The simplified pseudo-code example below illustrates the ease of which an application can programmatically acquire streaming video:

```
OpenVideo(“CameraLink_11”, “National Instruments:img0”, &myVideoID);
StartVideo(myVideoID);              // Start hardware image acquisition.
GrabFrame(myVideoID, &myBuffer);    // Obtain image,(and repeat if necessary).
StopVideo(myVideoID);               // Stop image acquisition.
CloseVideo(myVideoID);              // Release resources.
```

The programming model for Command & Control as well as Configuration follows the same style. Each of these interfaces and their defined functions are described below.

Part II – Discovery Interface

The discovery interface of ISC_Camera enables applications to discover and enumerate available hardware interfaces and devices for each interface.

Mandatory Discovery Functions

To export it’s available interfaces, implementation of the following functions is required of a given hardware interface (depending on whether that device interface supports Command-and-Control, Video, or Config, or potentially a combination thereof).

GetControlIF()

Returns the names of available command & control-capable interfaces on the system. You can call this function repeatedly to discover all available command & control interfaces.

Prototype:

```
isc_Error GetControlIF(BSTR * const IFName, const int index);
```

Parameters:

IFName:	A pointer to a BSTR to hold the discovered interface name.
index:	An index, starting from 0, that you should increment during each repeated call until the function returns eOutOfRange.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. eOutOfRange when there are no more interfaces available.

Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

To discover all available interfaces, call this function repeatedly while incrementing the value of index until eOutOfRange is returned.

GetVideoIF()

Returns the names of available video-capable interfaces on the system. You can call this function repeatedly to discover all available video interfaces.

Prototype:
```
isc_Error GetVideoIF(BSTR * const IFName, const int index);
```

Parameters:

IFName:	A pointer to a BSTR to hold the discovered interface name.
index:	An index, starting from 0, that you should increment during each repeated call until the function returns eOutOfRange.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. eOutOfRange when there are no more interfaces available.

Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

To discover all available interfaces, call this function repeatedly while incrementing the value of index until eOutOfRange is returned.

GetConfigIIF()

Returns the names of available configuration-capable interfaces on the system. You can call this function repeatedly to discover all available device configuration interfaces.

Prototype:
```
isc_Error GetConfigIF(BSTR * const IFName, const int index);
```

Parameters:

IFName:	A pointer to a BSTR to hold the discovered interface name.
index:	An index, starting from 0, that you should increment during each repeated call until the function returns eOutOfRange.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. eOutOfRange when there are no more interfaces available.

Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

To discover all available interfaces, call this function repeatedly while incrementing the value of index until eOutOfRange is returned.

GetIFDevice()

Returns the names of available devices for a given interface on the system. You can call this function repeatedly to discover all available devices.

Prototype:

```
isc_Error GetIFDevice(const BSTR IFName, BSTR * const deviceName,
            const int index );
```

Parameters:

IFName:	The name of the interface as received by a call to one of GetControlIF(), GetVideoIF(), or GetConfigIF() discovery function calls (or alternately, the name of a known interface).
deviceName:	A pointer to a BSTR to hold the discovered device name.
index:	An index, starting from 0, that you should increment during each repeated call until the function returns eOutOfRange.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. eOutOfRange when there are no more interfaces available.

Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

To discover all available devices, call this function repeatedly while incrementing the value of index until eOutOfRange is returned.

IsAvailable()

This convenience function does not have (or need) a hardware-specific implementation, but it allows an application to determine if a hardware-specific module has implemented API calls on a given interface before invoking the function. (Note: Use of this function is potentially costly – see remarks below).

Prototype:

```
int IsAvailable(const BSTR IFName, const BSTR FxnName)
```

Parameters:

FxnName: The name of the interface on which the availability is being tested.

FxnName: The name of the function of which the availability is being tested.

Return Values:

Non-zero (true) if the function exists, and zero (false) otherwise.

Remarks:

Use this function only for optional functions, as it creates unnecessary overhead to call this function for testing of *mandatory* functions. If any of the mandatory functions are missing, ISC_Camera will not load that hardware-specific module in the first place, (and therefore will not be available as a discoverable interface anyway).

Additionally, try to avoid calling this function repeatedly as a verification method. In such situations, a preferable method would be storing the result in a variable for future reference during the application session.

Example:

```
bool avail;

avail = (bool)IsAvailable(my_interface, “RefreshDeviceList”);
if (avail) {
      err = RefreshDeviceList();
}
```

Optional Discovery Functions

RefreshDeviceList()

Returns an updated device list for a given interface. It is necessary to call this function only if a change to the device list is expected.

Note: When this function is called for an interface, that same interface cannot be open for any type of access. Otherwise, ISC_Camera will return “eInvalidState” when this function is invoked.

Prototype:

```
isc_Error RefreshDeviceList(const BSTR IFName);
```

Parameters:

IFName: The name of the interface as received by a call to one of GetControlIF(), GetVideoIF(), or GetConfigIF() discovery function calls (or alternately, the name of a known interface).

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error.

eInvalidState if any of the interfaces of the device is open when the call is made.

Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use IsAvailable() to check its availability before invoking this function.

Part III – Video Interface

Mandatory Video Functions

Implementation of the following functions is required of a given hardware interface. ISC_Camera module will load and make available for discovery only the hardware modules that provide all of the following mandatory functions.

OpenVideo()

Opens a video communication channel to the specified device. This function returns an identifier for later use with all functions related to the chosen video device.

Prototype:
```
isc_Error OpenVideo(const BSTR IFName, const BSTR deviceName,
          int* const deviceId);
```

Parameters:

IFName:	The name of the interface as received by a call to the GetVideoIF() discovery function (or alternately, the name of a known interface).
deviceName:	The name of the device for which a video channel is to be opened. This name is received by a call to the GetIFDevice() discovery function (or alternately, the name of a known device).
deviceID:	A pointer to the variable in which to receive the interface identifier for later use.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

A call to this function is *required* before accessing any other video acquisition functions. Be sure to make a call to CloseVideo() when you are finished using the interface.

CloseVideo()

Closes access to a previously opened video communication interface.

Prototype:
```
isc_Error CloseVideo(const int deviceID);
```

Parameters:

deviceID:	Device id that was received by a call to OpenVideo().

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

To ensure proper system resource usage, a call to this function is required once for every successful OpenVideo() call.

StartVideo()

Commands the device to start asynchronous streaming of video data. The rate of acquisition and transmission is set by the existing hardware configuration (or by the underlying hardware-specific software).

Prototype:
```
isc_Error StartVideo(const int deviceID, const int channelID);
```

Parameters:

deviceID:	The interface id that was received by a call to OpenVideo().
channelID:	The video channel number of the device. Use 0 for devices that have only one video channel.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

Depending on the hardware platform, certain video parameters would be settable by calling functions such as SetFrameSize, SetFrameSkipCount, etc, or by using the appropriate Config interface functions to set the hardware configuration as whole.

StopVideo()

Commands the device to stop asynchronous streaming of video data on a previously started video stream.

Prototype:
```
isc_Error StopVideo(const int deviceID, const int channelID);
```

Parameters:

deviceID:	The id that was received by a call to OpenVideo().
channelID:	The video channel number of the device. Use 0 for devices that have only one video channel.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

To ensure proper resource usage, be sure to call this function for every successful StartVideo() call.

GrabFrame()

Requests one complete video frame video data from the given video channel.

Prototype:

```
isc_Error GrabFrame(const int deviceID, const int channelID,
          short *const buf);
```

Parameters:

deviceID:	The id that was received by a call to OpenVideo().
channelID:	The video channel number of the device. Use 0 for devices that have only one video channel.
buf:	A pointer to the user buffer to hold the image data.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

If StartVideo() **was** called before calling GrabFrame(), data acquisition will be *asynchronous* to requests from the application. On the other hand, if StartVideo() **was not** called first, data acquisition is *synchronous* to application's requests. Support for this feature will depend on hardware capability and the hardware-specific software module implementation.

GetFrameSize()

Reads the existing frame size of a given video device.

Prototype:

```
isc_Error GetFrameSize(const int deviceID, const int channelID,
          short *const rows , short *const cols);
```

Parameters:

deviceID:	The id that was received by a call to OpenVideo().
channelID:	The video channel number of the device. Use 0 for devices that have only one video channel.
rows:	A pointer to the variable to hold number of rows of the frame (i.e. frame height).
cols:	A pointer to the variable to hold number of columns of the frame (i.e. frame width).

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

SetFrameSize()

Writes to the device to set a desired video frame size.

Prototype:

```
isc_Error SetFrameSize(const int deviceID, const int channelID,
          const short rows, const short cols);
```

Parameters:

deviceID:	The id that was received by a call to OpenVideo().
channelID:	The video channel number of the device. Use 0 for devices that have only one video channel.
rows:	Number of rows of the frame (i.e. frame height).
cols:	Number of columns of the frame (i.e. frame width).

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Optional Video Functions

Implementation of each of the following functions is optional for a given hardware interface. Applications may use the discovery call "IsAvailable()" to check availability of each of the following calls before invoking them.

InitVideo()

Allows an application to initialize a video channel's communication devices, such as frame grabbers, to a known camera's default parameters, such as frame sizes, electrical link settings, etc. This function provides a convenient "single call" method to enable initialization of the frame grabber and communication equipment when the camera type is known.

Prototype:

```
isc_Error InitVideo(const int deviceID, const int channelID, const
          isc_CameraType cameraType);
```

Parameters:

deviceID:	The id that was received by a call to OpenVideo().
channelID:	The video channel number of the device. Use 0 for devices that have only one video channel.
cameraType:	A known camera type enumerated by the SDK. Please see the header file ISC_Types.h for enumeration of isc_CameraType.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use the discovery function IsAvailable() to check its availability before invoking this function.

GetVideoChannelCount()

Obtains the number of video channels supported by the selected device.

Prototype:

```
isc_Error GetVideoChannelCount(const int deviceId, int* const count);
```

Parameters:

deviceID:	The id that was received by a call to OpenVideo().
count:	A pointer to the variable to hold the number of channels available.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use the discovery function IsAvailable() to check its availability before invoking this function.

SetPixelDepth()

Sets the current video pixel depth for the selected device.

Prototype:

```
isc_Error SetPixelDepth( const int deviceId, const int channelID,
          isc_Depth iscDepth);
```

Parameters:

deviceID:	The id that was received by a call to OpenVideo().
channelID:	The video channel number of the device. Use 0 for devices that have only one video channel.
`iscDepth`:	Pixel depth as defined in the isc_Depth enumeration. Please see header file ISC_Types.h for the definition of isc_Depth.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use the discovery function IsAvailable() to check its availability before invoking this function.

GetPixelDepth()

Reads the current video pixel depth for the selected device.

Prototype:

```
isc_Error GetPixelDepth( const int deviceId, const int channelID,
          isc_Depth* const pIscDepth);
```

Parameters:

deviceID:	The id that was received by a call to OpenVideo().
channelID:	The video channel number of the device. Use 0 for devices that have only one video channel.
`pIscDepth`:	A pointer to the variable to hold the pixel depth as defined in the isc_Depth enumeration. Please see header file ISC_Types.h for the definition of isc_Depth.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use the discovery function IsAvailable() to check its availability before invoking this function.

SetFrameSkipCount()

Sets the number of frames from the camera to be skipped for each frame transmitted.

Prototype:

```
isc_Error SetFrameSkipCount(const int deviceId, const int channelID,
          const int skipCount);
```

Parameters:

deviceID:	The id that was received by a call to OpenVideo().

channelID: The video channel number of the device. Use 0 for devices that have only one video channel.

skipCount: Number of frames to be skipped for each transmitted frame.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

For frame grabbers that support this feature, setting the skip count is a method of reducing the bandwidth usage of the transmission medium by lowering the transmitted frame rate (independent of the camera's frame rate).

For example, a skip count of 0 will send all frames, and a skip count of 1 will send 50% of the frames, and a skip count of 2 will set the transmitted frames to be 1/3 the grab rate, etc.

In other words, Transmitted rate = (1 / (1 + count)) * Full frame rate.

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use the discovery function IsAvailable() to check its availability before invoking this function.

GetFrameSkipCount()

Obtains the frame grabber's current setting of number of video frames to be skipped for each frame transmitted.

Prototype:

```
isc_Error GetFrameSkipCount(const int deviceId, const int channelID,
          int * const skipCount);
```

Parameters:

deviceID: The id that was received by a call to OpenVideo().

channelID: The video channel number of the device. Use 0 for devices that have only one video channel.

skipCount: A pointer to the variable to hold the current skipCount.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

See remarks in SetFrameSkipCount() for an explanation of skipCount.

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use the discovery function IsAvailable() to check its availability before invoking this function.

GetOffsetX(), GetOffsetY()

Reads the current horizontal (X) or vertical (Y) offset defined for the frame grabber. The horizontal offset value indicates the number of pixel columns the frame grabber is to skip from the left hand edge of the incoming video frame. The vertical offset is the number of pixel rows the frame grabber is to skip from the top edge of the incoming video frame.

Prototype:

```
isc_Error GetOffsetX( const int deviceId, const int channelID,
          int * const x_offset);

isc_Error GetOffsetY( const int deviceId, const int channelID,
          int * const y_offset);
```

Parameters:

deviceID:	The id that was received by a call to OpenVideo().
channelID:	The video channel number of the device. Use 0 for devices that have only one video channel.
`x_offset`:	A pointer to the variable to hold the current horizontal offset.
`y_offset`:	A pointer to the variable to hold the current vertical offset.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use the discovery function IsAvailable() to check its availability before invoking this function.

SetOffsetX(), SetOffsetY()

Sets the current horizontal (X) or vertical (Y) offset defined for the frame grabber. The horizontal offset value indicates the number of pixel columns the frame grabber is to skip from the left hand edge of the incoming video frame. The vertical offset is the number of pixel rows the frame grabber is to skip from the top edge of the incoming video frame.

Prototype:

```
isc_Error SetOffsetX( const int deviceId, const int channelID,
          const int x_offset);

isc_Error SetOffsetY( const int deviceId, const int channelID,
          const int y_offset);
```

Parameters:

deviceID:	The id that was received by a call to OpenVideo().
channelID:	The video channel number of the device. Use 0 for devices that have only one video channel.
`x_offset`:	The new horizontal offset value.
`y_offset`:	The new vertical offset value.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use the discovery function IsAvailable() to check its availability before invoking this function.

Part IV – Command & Control Interface

Mandatory Command-and-Control Functions

Implementation of the following functions is required of a given hardware interface. ISC_Camera module will load and make available for discovery only the hardware modules that provide all of the following mandatory functions.

OpenControl()

Opens a command & control communication channel to the specified device. This function returns an identifier for later use with all command & control functions related to the chosen device.

Prototype:

```
isc_Error OpenControl(const BSTR IFName, const BSTR deviceName,
          int * const deviceId );
```

Parameters:

IFName:	The name of the interface as received by a call to the GetControlIF() discovery function (or alternately, the name of a known interface).
deviceName:	The name of the device for which a video channel is to be opened. This name is received by a call to the GetIFDevice() discovery function (or alternately, the name of a known device).
deviceID:	A pointer to the variable in which to receive the interface identifier for later use.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

A call to this function is *required* before accessing any other video acquisition functions. Be sure to make a call to CloseControl() when you are finished using the interface.

CloseControl()

Closes communications to a previously opened command & control communication interface.

Prototype:

```
isc_Error CloseControl(const int deviceId);
```

Parameters:

deviceID:	Device id that was received by a call to OpenControl().

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

To ensure proper resource usage, be sure to call this function for every successful OpenControl() call.

SetBaudRate()

Sets the baud rate of the command-and-control communication channel.

Prototype:
```
isc_Error SetBaudRate( const int deviceId, const int baudrate );
```

Parameters:

deviceID:	Device id that was received by a call to OpenControl().
baudrate:	Baud rate (actually data rate) in bits per second.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

GetBaudRate()

Obtains the baud rate (data rate) of the command-and-control communication channel.

Prototype:
```
isc_Error GetBaudRate( const int deviceId, int * const baudrate );
```

Parameters:

deviceID:	Device id that was received by a call to OpenControl().
baudrate:	A pointer to receive the baud rate (actually data rate) in bits per second.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

ReadControl()

Requests a given number of bytes of command & control data from a given device.

Prototype:
```
isc_Error ReadControl(const int deviceId,  char * const buf,
          int * const count );
```

Parameters:

deviceID:	Device id that was received by a call to OpenControl().
buf:	Pointer to the receive data buffer.
count:	Number of bytes requested. The function, however, may return with a less then the requested number of bytes filled into the data buffer. In that case, the count will be updated to indicate how many bytes were actually read.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

ReadControl() is intended to be a non-blocking function. However, its behavior depends on the implementation of the underlying hardware-specific module. In the case the function returns zero bytes, it is expected that the application will use a timer to schedule the next request appropriately.

WriteControl()

Writes the given number of bytes of command & control data to the device.

Prototype:
```
isc_Error WriteControl( const int deviceId,  char * const buf,
          int * const count );
```

Parameters:

deviceID:	Device id that was received by a call to OpenControl().
buf:	Pointer to the transmit data buffer.
count:	Number of bytes to be written.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error.
ePartialSuccess if a portion of the requested number of bytes were written.

Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

WriteControl() is designed to block until the requested number of bytes of data has been successfully written out to the device, or an error occurs. The function, however, may return with s less-than-the-requested number of bytes written to the device.

Optional Command-and-Control Functions

Implementation of each of the following functions is optional for a given hardware interface. Applications may use the discovery call “IsAvailable()” to check availability of each of the following calls before invoking them.

GetControlChannelCount()

Enables devices with multiple control channels to report the available number of channels.

Prototype:
```
isc_Error GetControlChannelCount(const int deviceId,
          int * const count);
```

Parameters:

deviceID:	Device id that was received by a call to OpenControl().

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error.
Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use the discovery function IsAvailable() to check its availability before invoking this function.

InitControl()

Allows an application to initialize a command and control channel’s communication devices to known camera default parameters, such as baud rate, a selectable UART, etc. This function provides a convenient “single call” method to initialization of communication equipment when the camera type is known.

Prototype:
```
isc_Error InitControl(const int deviceId, const int channelID,
          const int portID, const isc_CameraType cameraType);
```

Parameters:

deviceID:	The id that was received by a call to OpenControl().
channelID:	The control channel number of the device. Use 0 for devices that have only one control channel.
portID:	The port number of the device. Use 0 for devices that have only one port.
cameraType:	A known camera type enumerated by the SDK. Please see the header file ISC_Types.h for enumeration of isc_CameraType.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use the discovery function IsAvailable() to check its availability before invoking this function.

FlushControl()

Flushes the command and control channel for the given device.

Prototype:

```
isc_Error FlushControl( const int deviceId );
```

Parameters:

deviceID:	Device id that was received by a call to OpenControl().

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error.
Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use the discovery function IsAvailable() to check its availability before invoking this function.

Part V – Configuration Interface

The configuration interface allows obtaining and setting of both common and device-specific parameters. A configuration object is provided that contains:

i. A standard configuration (defined as isc_Config), and
ii. Device-specific information

inside a single object. The implementation of the hardware-specific module (at the bottom of the stack), and the application (at the top of the stack) are responsible for populating and using the contents of the configuration object. ISC_Camera will simply pass the same object pointer between the application and the device-specific module. For example, a device-specific configuration called myConfig may be created similar to the following model:

```
typedef struct {
     isc_Config std;            // standard settings
     int        myVariables;    // device specific settings
} myConfig;
```

Once this definition is available to both the application and the hardware-specific module of the SDK, it will be possible for the application to obtain and set extensive hardware configuration parameters.

Please see the header file ISC_Types.h for the definition of `isc_Config`.

Mandatory Config Functions

Implementation of the following functions is required of a given hardware interface. ISC_Camera module will load and make available for discovery only the hardware modules that provide all of the following mandatory functions.

OpenConfig()

Opens a communication channel to the specified device to read or write its configuration data. This function returns an identifier for later use with all functions related to the chosen device.

Prototype:
```
isc_Error OpenConfig(const BSTR IFName, const BSTR deviceName,
          int * const deviceId);
```

Parameters:

IFName:	The name of the interface as received by a call to the GetConfigIF() discovery function (or alternately, the name of a known interface).
deviceName:	The name of the device for which a video channel is to be opened. This name is received by a call to the GetIFDevice() discovery function (or alternately, the name of a known device).
deviceID:	A pointer to the variable in which to receive the interface identifier for later use.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

A call to this function is *required* before accessing any other video acquisition functions. Be sure to make a call to CloseConfig() when you are finished using the interface.

CloseConfig()

Closes communications to a previously opened configuration interface.

Prototype:
```
isc_Error CloseConfig(const int deviceId);
```

Parameters:

deviceID:	The id that was received by a call to OpenConfig().

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

To ensure proper system resource usage, a call to this function is required once for every successful OpenConfig() call.

GetConfigObjSize()

Obtains the size, in bytes, of the binary configuration object that could be retrieved by calling GetConfig() next. An application can cast the object pointer to the appropriate struct definition to access elements within the object.

Prototype:
```
isc_Error GetConfigObjSize(const int deviceId, int * const size);
```

Parameters:

deviceID:	The id that was received by a call to OpenConfig().
size:	A pointer to the variable in which to receive the size.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

You can call this function before allocating memory to determine the needed size of memory inside the application.

GetConfig()

Reads a device-specific binary configuration data object into the provided buffer.

Prototype:
```
isc_Error GetConfig(const int deviceId, char * const buf,
          int * const size );
```

Parameters:

deviceID:	A pointer to the variable in which to receive the interface identifier for later use.
buf:	Pointer to the configuration receive buffer.
size:	Size of the buffer.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Some implementations may return ePartialConfig if only a portion of the configuration was retrievable.

Remarks:
The binary configuration object may contain both generic and device-specific data. Please refer to the "isc_Config" structure in ISC_Types.h for device-independent data format, and other device-specific header files for device-specific data format.

Note: It may also be more convenient to use ExportConfig()/ImportConfig() functions for this same purpose, but that involves file I/O, which may or may not be an issue depending on your host platform and device-specific SDK functionality.

SetConfig()

Writes a device-specific binary configuration object provided in the buffer to the specified device.

Prototype:
```
isc_Error SetConfig( const int deviceId, char * const buf,
          const int size );
```

Parameters:

deviceID:	A pointer to the variable in which to receive the interface identifier for later use.
buf:	Pointer to the configuration buffer.
size:	Size of the buffer.

Return Values:
Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:
The binary configuration object may contain both generic and device-specific data. Please refer to the "isc_Config" structure in ISC_Types.h for device-independent data format, and other device-specific header files for device-specific data format.

Note: It may also be more convenient to use ExportConfig()/ImportConfig() functions for this same purpose, but that involves file I/O, which may or may not be an issue depending on your host platform and device-specific SDK functionality.

Optional Config Functions

Implementation of each of the following functions is optional for a given hardware interface. Applications may use the discovery call "IsAvailable()" to check availability of each of the following calls before invoking them.

InitConfig()

Allows an application to initialize the given configuration buffer to appropriate values for the specified camera. This function provides a convenient "single call" method to enable initialization of the configuration when the camera type is known.

Prototype:

```
isc_Error InitConfig(const int deviceId, const isc_CameraType cameraType);
```

Parameters:

deviceID:	The id that was received by a call to OpenControl().
cameraType:	A known camera type enumerated by the SDK. Please see the header file ISC_Types.h for enumeration of isc_CameraType.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use the discovery function IsAvailable() to check its availability before invoking this function.

ExportConfig()

Allows an application to instruct the hardware-specific module to export the device's current configuration to a file specified by the user. Since the activity is performed by the hardware-specific module, any file format appropriate to the hardware device (such as an XML, or binary file) can be chosen.

Prototype:

```
isc_Error ExportConfig(const int deviceId, const BSTR FilePath);
```

Parameters:

deviceID:	The id that was received by a call to OpenControl().
FilePath:	The full path to the configuration file to be exported.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use the discovery function IsAvailable() to check its availability before invoking this function.

ImportConfig()

Allows an application to instruct the hardware-specific module to import a user-specified device configuration from a file. The hardware specific-module is responsible for the interpretation and loading of the imported configuration.

Prototype:

```
isc_Error ExportConfig(const int deviceId, const BSTR FilePath);
```

Parameters:

deviceID:	The id that was received by a call to OpenControl().
FilePath:	The full path to the configuration file to be imported.

Return Values:

Zero if successful, otherwise returns an enumeration value defined in isc_Error. Please see the header file ISC_Types.h for enumeration of isc_Error.

Remarks:

Implementation of this function is optional for the hardware-specific module. Therefore, it may not be available on all interfaces and/or devices. You can use the discovery function IsAvailable() to check its availability before invoking this function.

Part VI – Appendixes

Appendix A – Interface Capabilities

Following table identifies currently implemented hardware interfaces and features:

Module	Cmnd & Control	Digital Video	Config	Notes
Ramp		√		Software simulation of a static ramp.
MovingRamp		√		Software simulation of moving ramp.
MSCOMM	√			Comms access via Microsoft COM ports (RS-232)
iPort	√	√	√	Direct access to firmware 3.x iPORTs.
iPORT2	√	√	√	Direct access to firmware 4.x iPORTs with iPORT runtime v2.2 or later.
IMAQ		√		National Instruments grabbers via IMAQ.
Firewire	√	√		Indigo/Pixelink Firewire grabber device.
USB	√			RS-232 communications via Cielo USB interface.
CameraLink10	√			Comms via AIA Camera Link 1.0 Matrox, I/O Industries boards.
CameraLink11	√			Comms via AIA Camera Link 1.1.
BFCameraLink	√	√		BitFlow Camera Link (and other) boards via BitFlow SDK.