TELEDYNE SCIENTIFIC & IMAGING
Everywhereyoulook™

API REFERENCE MANUAL

IPS API VERSION 2.1

TELEDYNE SCIENTIFIC & IMAGING
Everywhereyoulook™

TABLE OF CONTENTS

INTRODUCTION

This document describes the API (Application Programming Interface) for the Image Processing and Scientific Software Development Toolkit (IPS SDK). The IPS API is used by scientific and image processing applications for frame acquisition, camera configuration, and image processing functionality.

SYSTEM OVERVIEW

Figure 1 illustrates the IPS SDK software architecture. The SDK consists of a set of DLL's and associated library and header files that enable an application to acquire images, or scientific data, from an attached camera or sensor and to perform image processing using provided image processing functions[1].

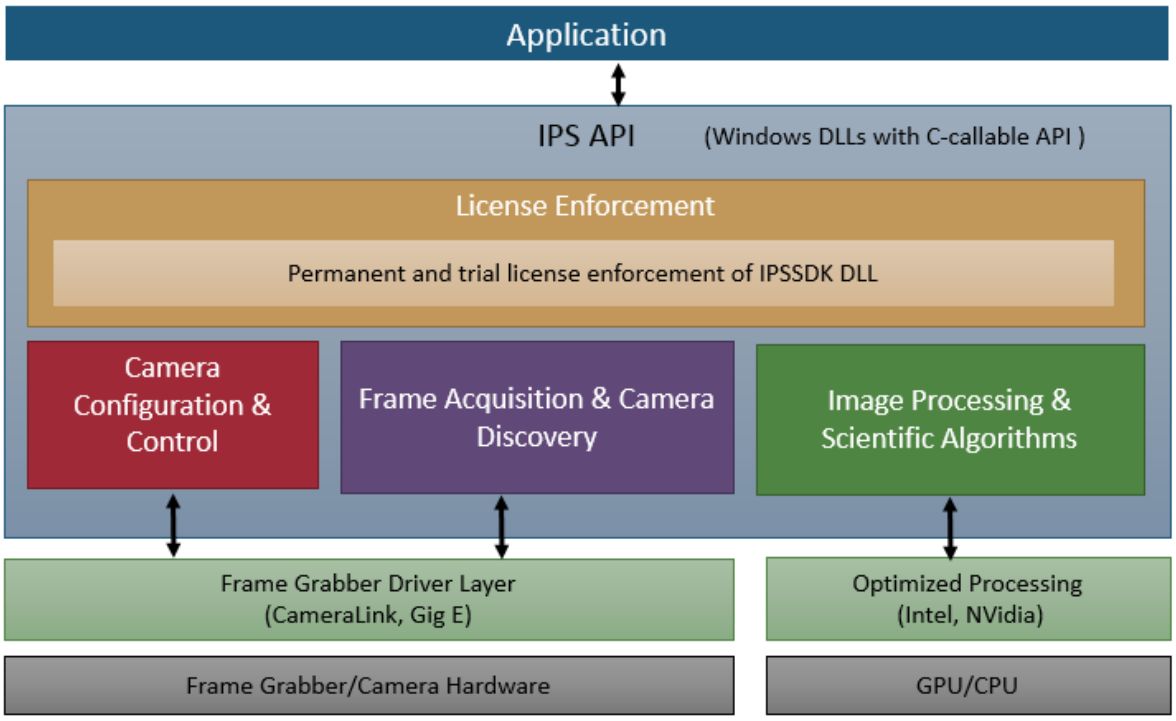

Figure 1. IPS API enables applications to access a common interface for frame acquisition and camera control. The IPS API also provides image processing functions such as common filters and primitive image operations as well as advanced algorithms such Teledyne Scientific's ADRC algorithm (*future release*).

[1] Image processing functions will be added in a future release of IPS SDK

API CONVENTIONS

C Calling Convention

IPS API functions use the C calling convention (*__cdecl*). This convention is the default for C/C++ programs compiled with the Microsoft Visual C++ compiler. In addition, if a C++ compiler is detected, the function names are declared with *extern "C",* to prevent the compiler from decorating the C function names. The IPS API uses POD (plain old data) data types. This allows the API functions to be called from a variety of programming languages and environments.

Memory Allocation

The calling application is required to allocate any input or output buffers required by the API. Pointers to these buffers must remain valid for the duration of the IPS operation. For example, a pointer to the ring-buffer provided to the IPS_StartContinuousGrabbing function must be valid the entire time frame acquisition is occurring. The calling application is responsible for de-allocating buffers that are no longer needed by the API.

The frame buffer pointers passed to IPS_StartContinuousGrabbing and IPS_StartGrabbing must be virtual pointers and should be allocated using the Windows VirtualAlloc function. Memory allocated for frame buffers will be paged locked by the driver, so it is important they be allocated using the Windows VirtualAlloc function. To make memory allocation easier, a helper class called VMemory is provided. This class will handle virtual memory allocation and will automatically free the virtual memory when the object is no longer used (i.e. when it is out of scope). See ***Acquiring Data*** for an example on using VMemory to allocate a frame buffer.

Return Codes (No Exceptions)

IPS API functions return status codes and do not throw exceptions. Errors codes are less than zero, warnings are greater than zero. A status code of zero indicates success. See ***Status Codes*** for a detailed listing of status codes.

Treatment of strings

All strings passed to IPS API functions are expected to contain UTF-8 encoded characters and to be terminated by the NULL character (i.e. a byte having a value of zero). Likewise, any string returned by the API will be UTF-8 encoded, NULL terminated strings.

When allocating memory for returned strings, it is imperative to allocate enough memory to include the NULL terminating character. Where applicable, IPS functions provide "IPS_MAX-*" constants for allocating string buffer memory. For example, IPS_MAX_DIAGNOSTIC_STRING_BYTES defines the maximum size necessary to allocate memory for the IPS diagnostic functions. These constants include the NULL terminator byte.

UTF-8 was chosen as an encoding standard because it supports multi-byte languages[2] while providing native support for ASCII (US-ASCII) character strings. For US-ASCII strings, no additional encoding/decoding is required since ASCII is a subset of UTF-8.

HEADER FILES, LIBRARIES, AND REDISTRIBUTABLES

The IPS SDK installation package will install the required IPS header files, libraries, and DLLs to the location chosen by the user when running the installer. Both x86 and x64 DLLs and .lib files are included in the installation package. DLL Files can be found under the *<Install Dir>\Bin* directory. The DLL import libraries can be found under the *<Install Dir>\Lib* directory. Header files can be found in the <Install Dir>\Include directory.

The IPS libraries require the Visual Studio 2010 SP1 runtime DLLs. The IPS SDK installer will copy the Visual Studio runtime installers (both x86 and x64) to the *<Install Dir>\Redistributables* directory.

SYSTEM REQUIREMENTS

The IPS SDK currently supports Windows 7, Windows 8, and Windows 8.1. Both x86 and native x64 libraries are available.

For frame acquisition, Dalsa Sapera LT 7.5 or newer is required. Both GigE Vision and Camera Link are supported by the IPS SDK. The IPS SDK installation package does *NOT* install Dalsa Sapera and the associated frame grabber drivers.

FUNCTION REFERENCE

This section provides a description of all functions available to the application developer. These functions are organized into the following categories:

- Capture Source Discovery

  Provides functions for enumerating all capture sources that are available for frame acquisition.

- Initialization and Cleanup

  Provides functions for initializing and de-initializing a capture source.

- Configuration and Control

  Provides functions for setting and retrieving camera and frame grabber parameters and for controlling the operation of the attached camera.

- Frame Acquisition

[2] The API currently supports UTF-8 strings with ASCII characters only.

Provides functions for acquisition of camera data.

- Utility

  Provides utility and diagnostic functions.

Capture Source Discovery

IPS_GetCaptureSourceCount

Retrieves a count of the number of available capture sources having the specified device type.

Syntax

```
int32_t IPS_GetCaptureSourceCount(
  int32_t    device_type,
  uint32_t * num_capture_sources_found
);
```

Parameters

device_type [in]
The type of device to retrieve: IPS_DEVICE_TYPE_CAMERALINK or IPS_DEVICE_TYPE_GIGE

num_capture_sources_found [out]
The number of capture sources found that match the specified device type.

Return value

Returns the status code. Returns IPS_SUCCESS if successful. Status code will be < 0 if an error occurs. See *Status Codes*.

Remarks

This function should be called before calling IPS_GetCaptureSource to determine how many capture sources are available. See ***Capture Source Identifier Format*** for information on capture source identifiers.

Examples

See *Initializing and Deintializing a Capture Source*.

See Also

IPS_GetCaptureSource

IPS_GetCaptureSource

Returns a capture source string which is used to identify a combination of the camera and frame grabber.

Syntax

```
int32_t IPS_GetCaptureSource(
  int32_t  device_type,
  uint32_t capture_source_index,
  char *   p_capture_source,
  uint32_t capture_source_size_bytes,
  char *   p_capture_source_descr,
  uint32_t capture_source_descr_bytes
);
```

Parameters

device_type [in]

The type of device to retrieve: IPS_DEVICE_TYPE_CAMERALINK or IPS_DEVICE_TYPE_GIGE

capture_source_index [in]

The index for retrieving the capture source identifier and the capture source description. The index must be >= 0 and less than the number of capture sources returned by IPS_GetCaptureSourceCount.

p_capture_source [out]

Points to a buffer that, upon return, will contain a null-terminated UTF-8 encoded character string containing the capture source identifier. The memory for this string must be allocated by the caller and must be large enough to store the null terminating character. The size of the allocated memory must be provided in the *capture_source_size_bytes* parameter. See *Capture Source Identifier Format* for information on capture source identifiers.

capture_source_size_bytes [in]

The size, in bytes, of the buffer pointed to by the p_capture_source parameter. IPS_MAX_CAPTURE_SOURCE_BYTES should be used to allocate a character array large enough to store the capture source identifier.

p_capture_source_descr [out]

Points to a buffer that, upon return, will contain a null-terminated UTF-8 encoded character string containing a description of the capture source. The memory for this string must be allocated by the caller and must be large enough to store the null terminating character. The size of the allocated memory must be provided in the *capture_source_descr_bytes* parameter. This parameter is optional and may be NULL.

capture_source_descr_bytes [in]

The size, in bytes, of the buffer pointed to by the *p_capture_source_descr* parameter. IPS_MAX_CAPTURE_SOURCE_BYTES should be used to allocate a character array large enough to store the capture source identifier.

Return value

Returns the status code. Returns IPS_SUCCESS if successful. Status code will be < 0 if an error occurs. See *Status Codes*.

Remarks

Call IPS_GetCaptureSourceCount to determine how many capture sources are available. The capture source identifier (i.e. p_capture_source) can be passed to IPS_InitAcq to obtain a handle that is required for the frame acquisition and configuration functions.

Examples

See *Initializing and Deintializing a Capture Source*.

See Also

IPS_GetCaptureSourceCount, IPS_InitAcq

Initialization and Cleanup

IPS_InitAcq

Allocates a new resource handle for accessing frame acquisition and camera/frame grabber configuration functions.

Syntax

```
int32_t IPS_InitAcq(
  uint32_t     camera_id,
  const char * p_capture_source,
  const char * p_xml_config,
  const char * p_license_file_path,
  HANDLE_IPS * p_handle
);
```

Parameters

camera_id [in]

Identifies the camera when sending custom camera configuration and control messages to a specific camera model over a serial communications link. CAM_ID_UNSPECIFIED can be specified if the camera does not require custom configuration over a serial link. See *Camera Identifiers*.

p_capture_source [in]

Points to a null-terminated UTF-8 encoded character string containing the capture source identifier. See *Capture Source Identifier Format* for information on capture source identifiers.

p_xml_config [in]

Points to a null-terminated UTF-8 encoded string containing XML configuration data for the camera or frame grabber. This is an OPTIONAL parameter. This parameter will be ignored if a NULL pointer is passed. See

Name	ID	Description
IPS_IMG_FORMAT_MONO8	0	8-bit monochrome
IPS_IMG_FORMAT_MONO10	10	10-bit monochrome
IPS_IMG_FORMAT_MONO16	1	16-bit monochrome
IPS_IMG_FORMAT_MONO24	2	24-bit monochrome
IPS_IMG_FORMAT_MONO32	3	32-bit monochrome
IPS_IMG_FORMAT_MONO64	4	64-bit monochrome
IPS_IMG_FORMAT_RGB5551	5	16 bits per pixel. 2 bytes containing R, G, B channels – 5 bits per channel plus a single bit alpha channel
IPS_IMG_FORMAT_RGB565	6	16 bits per pixel. 5 bits for R, 6 bits for G, and 5 bits fro B
IPS_IMG_FORMAT_RGB888	7	24-bits per pixel. 8 bits for R, 8 bits for G, 8 bits for B.
IPS_IMG_FORMAT_RGBR888	8	24-bits per pixel. 8 bits for R, 8 bits for G, 8 bits for B. Reversed order - R is stored first.
IPS_IMG_FORMAT_RGB8888	9	32-bit RGBA

XML Configuration Format.

p_license_file_path [in]

Points to a null-terminated UTF-8 encoded string containing a file path to a license file. A NULL pointer may be provided if a license is not required - e.g. if acquiring images on a camera that does not require license enforcement.

p_handle [out]

Upon return, will contain a handle resource. This handle uniquely identifies an instance of the capture source and is required for the frame acquisition, configuration, and utility functions.

Return value

Returns the status code. Returns IPS_SUCCESS if successful. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

This call will allocate a new resource handle for the specified capture source. This handle is required by most IPS API functions. It is *vitally* important to call IPS_DeinitAcq when finished with the returned handle. Failure to call IPS_DeinitAcq may result in the capture source becoming unavailable the next time the application is executed.

Examples

See *Initializing and Deintializing a Capture Source*.

See Also

IPS_DeinitAcq, IPS_GetCaptureSource

IPS_DeinitAcq

Deallocates and reclaims camera and frame grabber resources associated with the specified handle.

Syntax

```
int32_t IPS_DeinitAcq(const HANDLE_IPS handle);
```

Parameters

handle [in]

A resource handle associated with a capture source. Identifies a capture source whose resources will be deinitialized.

Return value

Returns the status code. Returns IPS_SUCCESS if successful. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

It is *vitally* important to call this function for each handle that is returned by IPS_InitAcq.

Examples

See *Initializing and Deintializing a Capture Source*.

See Also

IPS_InitAcq

IPS_SetInt32Param

Sets a configuration/control parameter to the specified integer value.

Syntax

```
int32_t IPS_SetInt32Param(
  const HANDLE_IPS handle,
  uint32_t         param_id,
  int32_t          value
);
```

Parameters

handle [in]

A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

param_id [in]

Identifies the configuration/control parameter. See *Configuration and Control Parameter Identifiers.*

value [in]

The 32-bit integer value to set.

Return value

Returns the status code. Returns IPS_SUCCESS if successful. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

Use this function to set the configuration/control parameter to the specified int32_t value. See the *Configuration and Control Parameter Identifiers* section to find the appropriate parameter ID and type.

Examples

See *Configuring a Capture Source*.

See Also

IPS_GetInt32Param

IPS_SetDoubleParam

Sets a configuration/control parameter to the specified double precision floating point value.

Syntax

```
int32_t IPS_SetDoubleParam(
  const HANDLE_IPS handle,
  uint32_t         param_id,
  double           value
);
```

Parameters

handle [in]
A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

param_id [in]
Identifies the configuration/control parameter. See *Configuration and Control Parameter Identifiers.*

value [in]
The double precision floating point value to set.

Return value

Returns the status code. Returns IPS_SUCCESS if successful. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

Use this function to set the configuration/control parameter to the specified double value. See the *Configuration and Control Parameter Identifiers* section to find the appropriate parameter ID and type.

Examples

See *Configuring a Capture Source*.

See Also

IPS_GetDoubleParam

IPS_GetInt32Param

Gets the integer value of the specified configuration/control parameter.

Syntax

```
int32_t IPS_GetInt32Param(
  const HANDLE_IPS handle,
  uint32_t         param_id,
  int32_t *        p_value
);
```

Parameters

handle [in]

A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

param_id [in]

Identifies the configuration/control parameter. See *Configuration and Control Parameter Identifiers.*

p_value [out]

Will store the retrieved int32_t value

Return value

Returns the status code. Returns IPS_SUCCESS if successful. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

Use this function to get the int32_t value associated with the specified configuration/control parameter. See the *Configuration and Control Parameter Identifiers* section to find the appropriate parameter ID and type.

Examples

See *Configuring a Capture Source*.

See Also

IPS_SetInt32Param

IPS_GetDoubleParam

Gets the double precision floating point value of the specified configuration/control parameter.

Syntax

```
int32_t IPS_GetDoubleParam(
  const HANDLE_IPS handle,
  uint32_t         param_id,
  double *         p_value
);
```

Parameters

handle [in]

A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

param_id [in]

Identifies the configuration/control parameter. See *Configuration and Control Parameter Identifiers.*

p_value [out]

Will store the retrieved double value

Return value

Returns the status code. Returns IPS_SUCCESS if successful. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

Use this function to get the double precision floating point value associated with the specified configuration/control parameter. See the *Configuration and Control Parameter Identifiers* section to find the appropriate parameter ID and type.

Examples

See *Configuring a Capture Source*.

See Also

IPS_SetDoubleParam

IPS_StartContinuousGrabbing

Starts aquiriing images from the capture source.

Syntax

```
int32_t IPS_StartContinuousGrabbing(
  const HANDLE_IPS handle,
  uint8_t *        p_ring_buffer,
  uint64_t         buffer_size_bytes
);
```

Parameters

handle [in]

A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

p_ring_buffer [in]

A buffer that has been allocated by the caller and will be used to store frame data. The buffer will be treated as a ring-buffer. This pointer is expected to be a Windows virtual pointer and must be allocated using VirtualAlloc (see remarks below).

buffer_size_bytes [in]

The size of the ring buffer in bytes

Return value

Returns the status code. Returns IPS_SUCCESS if successful. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

This function will start frame acquisition in continuous mode. Frame acquisition will continue until the frame acquisition is stopped using IPS_StopGrabbing, IPS_WaitFrame, or IPS_WaitFrame2 functions. The wait functions (i.e. IPS_WaitFrame and IPS_WaitFrame2) will only stop frame capture if the functions are called with the *pause* flag set to 1.

When the frame acquisition software fills the entire ring buffer with frames it will wrap around and begin overwriting the oldest frames with the newest frames.

The ring buffer must be allocated using VirtualAlloc. A helper class called VMemory is provided to simplify management of a virtual buffer. See *Acquiring Data* for an example on using VMemory to allocate the ring buffer.

The ring buffer need not be allocated to fit an integral number of frames. The frame acquisition software will divide the buffer size by the frame size and will leave the remaining portion of the buffer unused.

The ring buffer must be at least large enough for one frame, otherwise it will return IPS_ERROR_FRAME_GRABBING_USER_BUFFER_INSUFFICIENT.

Frame acquisition must stopped before calling IPS_StartContinuousGrabbing. To stop frame grabbing call IPS_StopGrabbing or call IPS_WaitFrame with the *pause* flag set to 1. If IPS_StartContinuousGrabbing is called while frame acquisition is in progress, IPS_WARNING_FRAME_GRABBING_IN_PROGRESS is returned and the function call has no effect.

The main difference between IPS_StartContinuousGrabbing and IPS_StartGrabbing is that frame acquisition started using IPS_StartGrabbing will be automatically stopped when the specified number of frames have been acquired. Frame acquisition started using IPS_StartContinuousGrabbing will not automatically stop.

Examples

See *Acquiring Data*.

See Also

IPS_StopGrabbing, IPS_StartGrabbing

IPS_StartGrabbing

Starts aquiriing a specified number of frames from the capture source. When the specified number of frames are captured, frame acquisition will be automatically stopped.

Syntax

```
int32_t IPS_StartGrabbing(
  const HANDLE_IPS handle,
  uint64_t         num_frames,
  uint8_t *        p_buffer,
  uint64_t         buffer_size_bytes,
  BOOL_IPS         allow_wrap
);
```

Parameters

handle [in]

A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

num_frames [in]

The number of frames to grab. Must be > 0.

p_buffer [in]

A buffer that has been allocated by the caller and will be used to store frame data. The buffer will be treated as a ring-buffer if *allow_wrap* is set to 1 and the requested number of frames will not fit in the provided buffer (i.e. if *buffer_size_bytes* < *num_frames**frame_byte_size). This pointer is expected to be a Windows virtual pointer and must be allocated using VirtualAlloc (see remarks below).

buffer_size_bytes [in]

The size of the buffer in bytes.

allow_wrap [in]

A flag that determines whether or not the buffer is to be treated as a ring buffer. If set to 1 and the buffer is full, the frame acquisition will automatically wrap around and begin replacing the oldest frames with the newest frames. If allow_wrap is set to 0 and the buffer size is insufficient to store the requested frames then an error will be returned.

Return value

Returns the status code. Returns IPS_SUCCESS if successful. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

This function will start frame acquisition such that frame acquisition will automatically stop once the specified number of frames have been acquired. Alternatively, frame acquisition can be stopped by the application using IPS_StopGrabbing, IPS_WaitFrame, or IPS_WaitFrame2 functions. The wait functions (i.e. IPS_WaitFrame and IPS_WaitFrame2) will only stop frame capture if the functions are called with the *pause* flag set to 1.

If allow_wrap is set to 1 and the buffer is not large enough to store all frames, frame acquisition will wrap around and begin overwriting the oldest frames with the newest frames.

The buffer must be allocated using VirtualAlloc. A helper class called VMemory is provided to simplify management of a virtual buffer. See *Acquiring Data* for an example on using VMemory to allocate the buffer.

The buffer need not be allocated to fit an integral number of frames. The frame acquisition software will divide the buffer size by the frame size and will leave the remaining portion of the buffer unused.

The buffer must be at least large enough for one frame, otherwise the function will return IPS_ERROR_FRAME_GRABBING_USER_BUFFER_INSUFFICIENT.

Frame acquisition must be stopped before calling IPS_StartGrabbing. To stop frame grabbing call IPS_StopGrabbing or IPS_WaitFrame with the *pause* flag set to 1. If IPS_StartGrabbing is called while frame acquisition is in progress, IPS_WARNING_FRAME_GRABBING_IN_PROGRESS is returned and the function call has no effect.

The main difference between IPS_StartGrabbing and IPS_StartContinuousGrabbing is that frame acquisition started using IPS_StartGrabbing will be automatically stopped when the specified number of frames have been acquired. Frame acquisition started using IPS_StartContinuousGrabbing will not automatically stop.

Examples

See *Acquiring Data*.

See Also

IPS_StopGrabbing, IPS_StartContinuousGrabbing

IPS_WaitFrame

Waits for the desired frame number to be acquired or a timeout to occur. If the desired frame was acquired successfully, a pointer to the frame is returned to the caller.

Syntax

```
int32_t IPS_WaitFrame(
  const HANDLE_IPS handle,
  uint64_t         desired_frame_number,
  int32_t          timeout_ms,
  BOOL_IPS         pause,
  uint8_t **       pp_frame,
  uint64_t *       p_frame_number
);
```

Parameters

handle [in]

A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

desired_frame_number [in]

The frame number of the desired frame. Frame numbers start at 1 and increment for each subsequently acquired frame. To wait for the next available frame pass 0 (i.e. IPS_GET_NEXT_FRAME) for desired_frame_number.

timeout_ms [in]

The duration of time to wait for the desired frame to be acquired, in milliseconds. Pass IPS_TIMEOUT_WAIT_FOREVER (i.e. -1) for an infinite wait.

pause [in]

Set to 1 to automatically pause frame acquisition before returning. If 0 is passed, frame acquisition will not be paused (see Remarks below).

pp_frame [out]

If the desired frame was acquired successfully, this parameter will contain a pointer to the desired frame data.

p_frame_number [out]

If the desired frame was acquired successfully, this parameter will contain the frame number associated with the frame data.

Return value

Returns the status code. Returns IPS_SUCCESS if successful. Returns IPS_WARNING_TIMEOUT if a timeout occurs. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

If the desired frame is successfully acquired, **pp_frame* will point to the frame data of the desired frame and **p_frame_number* will be set to the frame number of the desired frame.

If a timeout occurs, IPS_WARNING_TIMEOUT is returned, **pp_frame* is set to NULL, and **p_frame_number* is set to 0.

If the desired frame has been overwritten due to a ring buffer wrap condition, IPS_ERROR_FRAME_GRABBING_FRAME_OVERWRITTEN is returned, **pp_frame* is set to NULL, and **p_frame_number* is set to 0.

If *pause* is set to 1, frame acquisition will automatically pause before the function returns. This allows the application to process the frame data without the potential for overwrite, due to a ring buffer wrap condition. IPS_ResumeGrabbing should be called when finished processing the frame data in order to continue frame acquisition.

If *pause* is set to 0, care must be taken to avoid frames from being overridden due to a ring buffer wrap condition. The only time this is not a concern is if IPS_StartGrabbing is called with *allow_wrap* set to 0. In this situation the buffer is not treated as a ring buffer and it could be useful to call IPS_WaitFrame to check on progress of a long running frame acquisition process.

Examples

See *Acquiring Data*.

See Also

IPS_ResumeGrabbing, IPS_WaitFrame2

IPS_WaitFrame2

Waits for the desired frame number to be acquired or a timeout to occur. If the desired frame was acquired successfully, a pointer to the frame is returned to the caller. This function is nearly identicial to IPS_WaitFrame; the only difference is IPS_WaitFrame2 returns the most recent frame in addition to the desired frame.

Syntax

```
int32_t IPS_WaitFrame2(
  const HANDLE_IPS handle,
  uint64_t         desired_frame_number,
  int32_t          timeout_ms,
  BOOL_IPS         pause,
  uint8_t **       pp_desired_frame,
  uint64_t *       p_desired_frame_number,
  uint8_t **       pp_most_recent_frame,
  uint64_t *       p_most_recent_frame_number
);
```

Parameters

handle [in]

A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

desired_frame_number [in]

The frame number of the desired frame. Frame numbers start at 1 and increment for each subsequently acquired frame. To wait for the next available frame pass 0 (i.e. IPS_GET_NEXT_FRAME) for desired_frame_number.

timeout_ms [in]

The duration of time to wait for the desired frame to be acquired, in milliseconds. Pass IPS_TIMEOUT_WAIT_FOREVER (i.e. -1) to for an infinite wait.

pause [in]

Set to 1 to automatically pause frame acquisition before returning. If 0 is passed, frame acquisition will not be paused (see Remarks below).

pp_desired_frame [out]

If the desired frame was acquired successfully, this parameter will contain a pointer to the desired frame data.

p_desired_frame_number [out]

If the desired frame was acquired successfully, this parameter will contain the frame number associated with the frame data.

pp_most_recent_frame [out]

If the desired frame was acquired successfully, this parameter will contain a pointer to the most recent frame data. Note: this may or may not point to the desired frame number (See Remarks below)

p_most_recent_frame_number [out]

If the desired frame was acquired successfully, this parameter will contain the frame number associated with the most recent frame data.

Return value

Returns the status code. Returns IPS_SUCCESS if successful. Returns IPS_WARNING_TIMEOUT if a timeout occurs. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

If the desired frame is successfully acquired, **pp_desired_frame* will point to the frame data of the desired frame and **p_desired_frame_number* will be set to the frame number of the desired frame.

If a timeout occurs, IPS_WARNING_TIMEOUT is returned, **pp_desired_frame* is set to NULL, and **p_desired_frame_number* is set to 0.

If the desired frame has been overwritten due to a ring buffer wrap condition, IPS_ERROR_FRAME_GRABBING_FRAME_OVERWRITTEN is returned, **pp_desired_frame* is set to NULL, and **p_desired_frame_number* is set to 0.

If *pause* is set to 1, frame acquisition will automatically pause before the function returns. This allows the application to process the frame data without the potential for overwrite, due to a ring buffer wrap condition. IPS_ResumeGrabbing should be called when finished processing the frame data in order to continue frame acquisition.

If *pause* is set to 0, care must be taken to avoid frames from being overridden due to a ring buffer wrap condition. The only time this is not a concern is if IPS_StartGrabbing is called with *allow_wrap* set to 0. In this situation the buffer is not treated as a ring buffer and it could be useful to call IPS_WaitFrame to check on progress of a long running frame acquisition process.

It is possible the desired frame is not the same as the most recent frame. If the desired frame is acquired then **pp_most_recent_frame* will point to the most recent frame and **p_most_recent_frame_number* will be set to the frame number of the most recent frame. If the desired frame is not successful acquired, **pp_most_recent_frame* will be set to NULL and **p_most_recent_frame_number* will be set to 0. If the desired frame is the most recent frame then **pp_most_recent_frame* will be the same as **pp_desired_frame* and **p_most_recent_frame_number* will be the same as **p_desired_frame_number*.

Examples

See *Acquiring Data*.

See Also

IPS_ResumeGrabbing

Resumes a previously stopped frame acquisition.

Syntax

```
int32_t IPS_ResumeGrabbing(const HANDLE_IPS handle);
```

Parameters

handle [in]

A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

Return value

Returns the status code. Returns IPS_SUCCESS if successful. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

Use this function to resume frame acquisition. Frame acquisition can be stopped by calling IPS_StopGrabbing or by calling one of the wait functions (i.e. IPS_WaitFrame or IPS_WaitFrame2) with the *pause* flag set to 1.

IPS_ResumeGrabbing is different than IPS_StartGrabbing or IPS_StartContinuousGrabbing. IPS_ResumeGrabbing will simply resume frame acquisition using the current buffer and frame position. Conversely, calling IPS_StartGrabbing or IPS_StartContinuousGrabbing will reset the frame acquisition and any data in the provided buffer will be overwritten from the beginning of the buffer.

Examples

See *Acquiring Data*.

See Also

IPS_StopGrabbing, IPS_WaitFrame, IPS_WaitFrame2

IPS_StopGrabbing

Stops frame acquisition.

Syntax

```
int32_t IPS_StopGrabbing(const HANDLE_IPS handle);
```

Parameters

handle [in]

A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

Return value

Returns the status code. Returns IPS_SUCCESS if successful. Returns IPS_WARNING_FRAME_GRABBING_NOT_STARTED if frame acquisition has already been stopped. Returns IPS_WARNING_FRAME_GRABBING_NOT_STARTED if the frame acquisition has not be started. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

Use this function to stop frame acquisition. Call IPS_ResumeGrabbing to resume frame acquisition or call one of the start functions (i.e. IPS_StartContinuousGrabbing or IPS_StartGrabbing) to restart the frame acquisition from the beginning of the buffer.

Examples

See *Acquiring Data*.

See Also

IPS_ResumeGrabbing, IPS_StartGrabbing, IPS_StartContinuousGrabbing

IPS_GetFrameSize

Returns the current frame size used for frame acquisition.

Syntax

```
int32_t IPS_GetFrameSize(
  const HANDLE_IPS handle,
  uint32_t * p_x_size,
  uint32_t * p_y_size
);
```

Parameters

handle [in]
A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

p_x_size [out]
Upon return, **p_x_size* will contain the size of the frame in the x dimension

p_y_size [out]
Upon return, **p_y_size* will contain the size of the frame in the x dimension

Return value

Returns the status code. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

Use this function to retrieve the frame x (width) and y (height) dimensions, in pixels.

See Also

IPS_SetFrameWindow, IPS_GetFrameWindowOffsets

IPS_GetFrameWindowOffsets

Returns the current frame window offsets used for frame acquisition.

Syntax

```
int32_t IPS_GetFrameWindowOffsets(
  const HANDLE_IPS handle,
  uint32_t *       p_x_offset,
  uint32_t *       p_y_offset
);
```

Parameters

handle [in]
A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

p_x_offset [out]
Upon return, **p_x_offset* will contain the offset of the frame in the x dimension

p_y_offset [out]
Upon return, **p_y_offset* will contain the offset of the frame in the x dimension

Return value

Returns the status code. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

Use this function to retrieve the frame offsets in the x and y dimensions, in pixels.

See Also

IPS_SetFrameWindow, IPS_GetFrameSize

IPS_SetFrameWindow

Sets the frame window offsets and size used for frame acquisition.

Syntax

```
int32_t IPS_SetFrameWindow(
  const HANDLE_IPS handle,
  uint32_t         x_offset,
  uint32_t         y_offset,
  uint32_t         x_size,
  uint32_t         y_size
);
```

Parameters

handle [in]
: A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

x_offset [in]
: The x offset of the frame

y_offset [in]
: The y offset of the frame

x_size [in]
: The size of the frame in the x dimension

y_size [in]
: The size of the frame in the y dimension

Return value

Returns the status code. Status code will be < 0 if an error occurs. See *Status Codes.*

Remarks

Use this function to set the frame acquisition window offsets and size.

Examples

See *Acquiring Data*.

See Also

IPS_GetFrameSize, IPS_GetFrameWindowOffsets

Utility

IPS_GetCameraDiagnostics

Returns a string containing camera diagnostic information.

Syntax

```
int32_t IPS_GetCameraDiagnostics(
  const HANDLE_IPS handle,
  char *           diagnostics_str_buffer,
  uint32_t         buffer_size_in_bytes,
  uint32_t *       p_buffer_size_out_bytes
);
```

Parameters

handle [in]

A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

diagnostics_str_buffer [out]

Points to a buffer that, upon return, will contain a null-terminated UTF-8 encoded character string consisting of camera diagnostic information. The memory for this string must be allocated by the caller and must be large enough to store the null terminating character. The size of the allocated memory must be provided in the *buffer_size_in_bytes* parameter.

buffer_size_in_bytes [in]

The size, in bytes, of the buffer pointed to by the *diagnostics_str_buffer* parameter. IPS_MAX_DIAGNOSTIC_STRING_BYTES should be used to allocate a character array large enough to store the diagnostic string.

p_buffer_size_out_bytes [out]

The size of the diagnostic data, in bytes (includes the null terminating character).

Return value

Returns the status code. Status code will be < 0 if an error occurs. Returns IPS_ERROR_BUFFER_INSUFFICIENT if the buffer is not large enough to store the diagnostic information. For other errors see *Status Codes.*

Remarks

This function returns a camera diagnostic string that can be used as a debugging aid. The buffer pointed to by *diagnostics_str_buffer* should be allocated using IPS_MAX_DIAGNOSTIC_STRING_BYTES.

Examples

See *Acquiring Data*.

IPS_GetFrameGrabberDiagnostics

Returns a string containing frame grabber diagnostic information.

Syntax

```
int32_t IPS_GetFrameGrabberDiagnostics(
  const HANDLE_IPS handle,
  char *           diagnostics_str_buffer,
  uint32_t         buffer_size_in_bytes,
  uint32_t *       p_buffer_size_out_bytes
);
```

Parameters

handle [in]

A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

diagnostics_str_buffer [out]

Points to a buffer that, upon return, will contain a null-terminated UTF-8 encoded character string consisting of frame grabber diagnostic information. The memory for this string must be allocated by the caller and must be large enough to store the null terminating character. The size of the allocated memory must be provided in the *buffer_size_in_bytes* parameter.

buffer_size_in_bytes [in]

The size, in bytes, of the buffer pointed to by the *diagnostics_str_buffer* parameter. IPS_MAX_DIAGNOSTIC_STRING_BYTES should be used to allocate a character array large enough to store the diagnostic string.

p_buffer_size_out_bytes [out]

The size of the diagnostic data, in bytes (includes the null terminating character).

Return value

Returns the status code. Status code will be < 0 if an error occurs. Returns IPS_ERROR_BUFFER_INSUFFICIENT if the buffer is not large enough to store the diagnostic information. For other errors see *Status Codes.*

Remarks

This function returns a frame grabber diagnostic string that can be used as a debugging aid. The buffer pointed to by *diagnostics_str_buffer* should be allocated using IPS_MAX_DIAGNOSTIC_STRING_BYTES.

Examples

See *Acquiring Data*.

IPS_CalculateIntegrationTime

Calculates the integration time corresponding to given values of the integration row and column ticks.

Syntax

```
int32_t IPS_CalculateIntegrationTime(
  const HANDLE_IPS handle,
  int row_ticks,
  int col_ticks,
  double * p_tm_seconds
)
```

Parameters

handle [in]
: A resource handle associated with a capture source. Call IPS_InitAcq to obtain a resource handle.

row_ticks [in]
: Timing ticks for a row of pixels.

col_ticks [in]
: Timing ticks for a column of pixels.

p_tm_seconds [out]
: The integration time, in seconds.

Return value

Returns the status code. Status code will be < 0 if an error occurs. Returns IPS_ERROR_BUFFER_INSUFFICIENT if the buffer is not sufficient large for the diagnostic information code. For other errors see *Status Codes.*

Remarks

Examples

See *Configuring a Capture Source*

Status Codes

The following is a table of the status codes returned by the IPS API functions. Warning status codes are > 0. Errors are < 0.

IPS_SUCCESS	0
IPS_WARNING_TIMEOUT	1
IPS_WARNING_FRAME_GRABBING_IN_PROGRESS	2
IPS_WARNING_FRAME_GRABBING_ALREADY_PAUSED	3
IPS_WARNING_FRAME_GRABBING_FINISHED	4
IPS_WARNING_FRAME_GRABBING_NOT_STARTED	5
IPS_ERROR_LIBRARY_NOT_LOADED	-1
IPS_ERROR_SOURCE_NOT_FOUND	-2
IPS_ERROR_SOURCE_NOT_SUPPORTED	-3
IPS_ERROR_SOURCE_INIT_FAILED	-4
IPS_ERROR_SOURCE_UNKNOWN_DATA_FORMAT	-5
IPS_ERROR_NOT_INTIALIZED	-6
IPS_ERROR_FRAME_GRABBING_FAILED	-7
IPS_ERROR_FRAME_GRABBING_USER_BUFFER_INVALID	-8
IPS_ERROR_BUFFER_INSUFFICIENT	-9
IPS_ERROR_FRAME_GRABBING_NUM_FRAMES_INVALID	-10
IPS_ERROR_FRAME_GRABBING_DESIRED_FRAME_NUM_INVALID	-11
IPS_ERROR_FRAME_GRABBING_NOT_STARTED	-12
IPS_ERROR_FRAME_GRABBING_FRAME_OVERWRITTEN	-13
IPS_ERROR_PARAM_SET_FAILED	-14
IPS_ERROR_PARAM_GET_FAILED	-15
IPS_ERROR_PARAM_UNKNOWN	-16
IPS_ERROR_REQUIRED_PARAM_NOT_SET	-17
IPS_ERROR_CAMERA_INIT_FAILED	-18
IPS_ERROR_CAMERA_NOT_INITIALIZED	-19
IPS_ERROR_INVALID_POINTER	-20
IPS_ERROR_XML_PARSE_FAILED	-21
IPS_ERROR_XML_CONFIG_FILE_NOT_FOUND	-22
IPS_ERROR_INVALID_HANDLE	-50
IPS_ERROR_LIC_ENF_INTERNAL_ERROR	-120
IPS_ERROR_LIC_ENF_HOSTID_MISMATCH	-121
IPS_ERROR_LIC_ENF_EXPIRED	-122
IPS_ERROR_LIC_ENF_CLOCK_TAMPERED	-123
IPS_ERROR_LIC_ENF_FEATURE_MISMATCH	-124
IPS_ERROR_LIC_ENF_WRONG_VERSION	-125
IPS_ERROR_LIC_ENF_ACCESS_DENIED	-126
IPS_ERROR_LIC_ENF_INVALID	-127
IPS_ERROR_LIC_MISSING	-128
IPS_ERROR_LIC_NOT_LICENSED	-129
IPS_ERROR_EXCEPTION	-1000

Camera Identifiers

The following table provides a list of supported camera identifiers. A camera identifier is provided to the IPS_InitAcq function.

Name	ID	Description
CAM_ID_NONE	0	*Future Release* : No camera is specified. Provide this value to IPS_InitAcq only using theimage processing API functions. If this ID is specified in the call to IPS_InitAcq, the camera configuration and frame acquisition functions should not be called.
CAM_ID_GENERIC	1	Provide this value to IPS_InitAcq to use a generic camera object. This ID should be called if the camera does not require custom configuration and control.
CAM_ID_SBF161	2	Specifies the Teledyne Nova Pour Filled LWIR camera built around the SBF 161 ROIC. Providing this value will enable configuration and control of the SBF 161 camera as well as frame acquisition via a Camera Link interface.

Jam Sync Modes

The following table provides a description of the possible jam sync modes. The jam sync mode determines how the camera is synchronized with external signals.

Name	ID	Description
IPS_JAM_SYNC_NONE	0	The camera will be free-running.
IPS_JAM_SYNC_TTL_OUT	1	The camera will be free-running, but it will ouput a TTL pulse at the start of every frame.
IPS_JAM_SYNC_TTL_IN	2	The camera will produce a frame each time it receives a TTL pulse. Otherwise, it will not produce a frame.
IPS_JAM_SYNC_RS422_IN	3	The camera will produce a frame each time it receives a signal through the RS422 port. Otherwise, it will not produce a frame.

Configuration and Control Parameter Identifiers

The following table provides a brief description of each configuration and control identifier. Use the type information to call the appropriate configuration function. For example, to retrieve a value for a parameter whose type is double, call IPS_GetDoubleParam. For identifiers containing the word "GET" call the corresponding IPS_GetDoubleParam or IPS_GetInt32Param. For identifiers containing the word "SET" or "WRITE" call the IPS_SetDoubleParam or IPS_SetInt32Param function.

Parameter Name	ID	Type	Description
IPS_GET_FRAME_RATE	100	int32_t	Retreives the expected frame rate (frames/second. This parameter is NOT the frame rate actually obtained, rather it is the current expected frame rate. The frame acquisition may use this rate to optimize the frame acquisition process. Use IPS_GET_GRAB_RATE to retrieve the estimated rate at which data frames are being acquired.
IPS_GET_INTEGRATION_TIME	101	double	Returns the current integration time (seconds). Uses the col and row ticks previously read from the camera to calculate integration time in seconds. Does not communicate with camera head.
IPS_WRITE_INTEGRATION_ROW_TICKS	102	int32_t	Sets the number of row ticks. Also recalculates the current integration time. Does not write the row ticks value to the camera head. Use IPS_WRITE_INTEGRATION_TIME to write the current row and col tick values to the camera head.
IPS_WRITE_INTEGRATION_COL_TICKS	103	int32_t	Sets the number of column ticks. Also recalculates the current integration time. Does not write the column ticks value to the camera head. Use IPS_WRITE_INTEGRATION_TIME to write the current row and col tick values to the camera head.
IPS_WRITE_INTEGRATION_TIME	104	int32_t	The integration time is determined by the number of integration row and column ticks stored on the camera head. This function writes the current row and column ticks to the camera head. Therfore, to change the integration time use IPS_WRITE_INTEGRATION_COL_TICKS and IPS_WRITE_INTEGRATION_ROW_TICKS

			prior to using this parameter. The int32_t value of this parameter is ignored.
IPS_READ_INTEGRATION_TIME	105	double	Reads the row and column integration tick times from the camera head and returns the calculated integration time (in seconds) based on the row and column tick values.
IPS_WRITE_JAM_SYNC	106	int32_t	Writes the jam sync mode to the camera head. The jam sync mode determines how the camera is synchronized with external signals. See *Jam Sync Modes.*
IPS_GET_COL_START	107	int32_t	Gets the focal plane column start position. Returns the current column start value – it does not communicate with the camera head.
IPS_GET_COL_END	108	int32_t	Gets the focal plane column end position. Returns the current column end value – it does not communicate with the camera head.
IPS_GET_ROW_START	109	int32_t	Gets the focal plane row start position. Returns the current row start value – it does not communicate with the camera head.
IPS_GET_ROW_END	110	int32_t	Gets the focal plane row end position. Returns the current row end value – it does not communicate with the camera head.
IPS_GET_COL_TICKS	111	int32_t	Get the number of timing column ticks. This function gets the number of timing column ticks needed to read one row of pixels. This number is related to the number of FPA columns and can be useful for diagnostics. Does not communicate with camera head.
IPS_GET_ROW_TICKS	112	int32_t	Get the number of timing row ticks. This function gets the number of timing row ticks needed to read one frame. This number is related to the number of FPA rows and can be useful for diagnostics. Does not communicate with camera head.
IPS_GET_FPA_TYPE	113	int32_t	Returns the FPA type of the current camera. Currently, only FPA type IPS_FPA_SBF161 is supported.
IPS_GET_SERIAL_NUM	114	int32_t	Gets the serial number previously read from the camera head. Does not communicate with the camera head.

IPS_WRITE_FRAME_COUNTER_ENABLED	115	int32_t	Sets the frame counter enabled bit on the camera head. When the frame counter is enabled, the first pixel of each frame will not correspond to an intensity, but to a counter that is incremented each frame (modulo 16384). This can be used to make sure no frames are being dropped. Set to 1 to enable frame counter mode, 0 to disable. Important: not all camera electronics support this mode. IPS_FPA_SBF161 does NOT support this mode.
IPS_READ_TEMP_FPA	116	double	Reads the FPA temperature (in Kelvin) from the camera head.
IPS_WRITE_COUNTER_TEST_MODE	117	int32_t	Writes the electronics test mode to the camera electronics. The test mode outputs a counter that ramps repeatedly from 0 to 16383. This is done without the help of the focalplane, so it can be used to test the operation of the camera electronics in isolation from the focalplane. Set to 1 to turn on test mode, set to 0 to turn off test mode.
IPS_GET_NUM_OUTPUTS	118	int32_t	Gets the number of A/D converters used by the current focal plane. This function does not communicate with the camera head.
IPS_GET_FPA_DETECTOR_BIAS	119	int32_t	Gets the current value of the camera head's FPA Detector Bias. The FPA detector bias value from 0 to 255. Some FPAs or electronics may support only 8, 16, or 32 different values. Returns -1 if the bias is not available. Important: The FPA detector bias is pre-set to give optimum performance and it should not generally need to be changed for InSb detectors.
IPS_WRITE_FPA_DETECTOR_BIAS	120	int32_t	Writes the camera head's FPA detector bias to the FPA. The provided value must be between 0 and 255, inclusive, though some FPAs or electronics may support only 8, 16, or 32 different values. Otherwise it is truncated to fit within this range. Important: The FPA detector bias is pre-set to give optimum performance and it should

			not generally need to be changed for InSb detectors.
IPS_READ_FRAME_COUNTER_ENABLED	121	int32_t	Reads the state of the frame counter enable bit from the camera head. When the frame counter is enabled, the first pixel of each frame will not correspond to an intensity, but to a counter that is incremented each frame (modulo 16384). This can be used to make sure no frames are being dropped. Returns 1 to if frame counter mode is enabled, 0 if frame counter mode is disabled. Important: not all camera electronics support this mode. IPS_FPA_SBF161 does NOT support this mode.
IPS_READ_JAM_SYNC	122	int32_t	Reads the current jam sync mode from the camera head. The jam sync mode determines how the camera is synchronized with external signals. See *Jam Sync Modes*.
IPS_READ_COUNTER_TEST_MODE	123	int32_t	Reads the electronics test mode bit from the camera electronics. The test mode outputs a counter that ramps repeatedly from 0 to 16383. This is done without the help of the focalplane, so it can be used to test the operation of the camera electronics in isolation from the focal plane. Returns 1 if enabled, 0 if disabled.
IPS_GET_INTEGRATION_ROW_TICKS	124	int32_t	Gets the number of integration row ticks previously read from the camera, which, together with the integration column ticks, determines the integration time. This function does not communicate with the camera head.
IPS_GET_INTEGRATION_COL_TICKS	125	int32_t	Gets the number of integration column ticks previously read from the camera, which, together with the integration row ticks, determines the integration time. This function does not communicate with the camera head.
IPS_SET_FRAME_RATE	126	int32_t	Sets the expected frame rate in integer frames/second. This parameter is optional and may be used by the frame acquisition software to optimize the frame acquistion process.

IPS_GET_GRAB_RATE	127	int32_t	Returns the estimated frame acquisition rate (in frames/second. This parameter can be used to guage the performance of the frame acquisition software.
IPS_SET_BIT_DEPTH	128	int32_t	This is the frame grabber bit depth, which may be different than the pixel bit depth of the image. Important: it is not recommended to change this setting as it depends on the frame grabber hardware in use.
IPS_GET_BIT_DEPTH	129	int32_t	Returns the bit depth used by the frame grabber hardware to acquire images. This is not necessarily the same is the pixel bit depth of the image.
IPS_WRITE_FAST_XFER_MODE	132	int32_t	Writes the camera head integration mode to Integrate While Read (i.e. fast transfer mode). Not all camera electronics support this mode while others only suuport fast transfer mode. IPS_FPA_SBF161 uses fast transfer mode and does NOT support changing to the slower Integrate Then Read mode.
IPS_GET_IMAGE_FORMAT	133	int32_t	Retrieves the current image format (e.g. IPS_IMG_FORMAT_MONO16 or IPS_IMG_FORMAT_MONO8). This parameter is used for image acquisition.
IPS_SET_IMAGE_FORMAT	134	int32_t	Sets the current image format (e.g. IPS_IMG_FORMAT_MONO16 or IPS_IMG_FORMAT_MONO8) This parameter is used for image acquisition.

IPS IMAGE FORMATS

Name	ID	Description
IPS_IMG_FORMAT_MONO8	0	8-bit monochrome
IPS_IMG_FORMAT_MONO10	10	10-bit monochrome
IPS_IMG_FORMAT_MONO16	1	16-bit monochrome
IPS_IMG_FORMAT_MONO24	2	24-bit monochrome
IPS_IMG_FORMAT_MONO32	3	32-bit monochrome
IPS_IMG_FORMAT_MONO64	4	64-bit monochrome
IPS_IMG_FORMAT_RGB5551	5	16 bits per pixel. 2 bytes containing R, G, B channels – 5 bits per channel plus a single bit alpha channel
IPS_IMG_FORMAT_RGB565	6	16 bits per pixel. 5 bits for R, 6 bits for G, and 5 bits fro B
IPS_IMG_FORMAT_RGB888	7	24-bits per pixel. 8 bits for R, 8 bits for G, 8 bits for B.
IPS_IMG_FORMAT_RGBR888	8	24-bits per pixel. 8 bits for R, 8 bits for G, 8 bits for B. Reversed order - R is stored first.
IPS_IMG_FORMAT_RGB8888	9	32-bit RGBA

XML CONFIGURATION FORMAT

The following shows the current XML configuration that may be provided to the IPS_InitAcq function. *Future releases may expand the definition to include additional configuration elements.* Notice the use of path expansion variables (see *Path Expansion Variables* for details)

```
<ips_config>
  <winir_inipath>$(IPS_SDK_DATA_DIR)\Config\winir.ini</winir_inipath>
  <ccf_path>$(IPS_SDK_DATA_DIR)\Config\sbf161_px4full_dv_1tap.ccf</ccf_path>
  <fg_window>
    <xoffset>0</xoffset>
    <yoffset>0</yoffset>
    <xsize>128</xsize>
    <ysize>128</ysize>
  </fg_window>
  <bytes_per_pixel>2</bytes_per_pixel>
  <pixel_depth>14</pixel_depth>
  <data_format>1</data_format> <!-- IPS_IMG_FORMAT_MONO16 -->
</ips_config>
```

Elements:

winir_inipath

The full path to the winir.ini file. Provides low level configuration to the SBF161 camera. This parameters is specific to the SBF161 camera and may not apply to other camera types. If this path is not provided, the system will first look in the current working directory for the winir.ini file and then will examine the WINIRDIR environment variable for the file path. The winir.ini data file maintains the same format as used by the SBF API and the WinIR program (developed by Santa Barbara Focalplane).

ccf_path

The full path to the Dalsa Sapera CCF configuration file.

fg_window

Provides the default settings for the frame grabber window (offsets and size). If this parameter is not provided the frame grabber window settings will be read from the current camera configuration.

bytes_per_pixel

Provides the default settings for the bytes per pixel. If this parameters is not provided, bytes per pixel will be inferred from the pixel depth.

pixel_depth

Provides the default settings for the pixel depth. If this parameter is not provided the pixel depth will be read from the current camera configuration.

data_format

Provides the desired image format. Must be one of the IPS image format values (see *IPS Image Formats*. If this parameters is not provided the format will be read from the current camera configuration.

PATH EXPANSION VARIABLES

File paths passed to the API may contain special path expansion variables to allow paths to by dynamically constructed using the local machine environment variables and settings. A path expansion variable has the form: $(VARIABLE_NAME). Path expansion variables are case insensitive. Any trailing separator characters will be removed from the expanded path variable so be sure to include a separator character (i.e. '\' or '/') after the expansion variable. For example, do this:
`"$(MyDocuments)\some_file.ccf"`, not this: `"$(MyDocuments)some_file.ccf"`.

The current set of supported variables are listed in the following table.

Name	Description
`$(MyDocuments)`	Expands to the current user's "My Documents" directory.
`$(WinIRDir)`	Expands to the current WINIRDIR environment variable setting.
`$(SaperaDir)`	Expands to the current SAPERADIR environment variable setting.
`$(IPS_SDK_DIR)`	Expands to the current IPS_SDK_DIR environment variable setting. The installer sets this environment variable to the install directory.
`$(IPS_SDK_DATA_DIR)`	Expands to the current IPS_SDK_DATA_DIR environment variable setting. The installer sets this environment variable to : "$(MyDocuments)\Teledyne Scientific & Imaging\IPS SDK"

CAPTURE SOURCE IDENTIFIER FORMAT

A capture source identifier contains essential frame acquisition parameters required to initialize the lower level frame acquisition software. The IPS capture source discovery functions (*see Capture Source Discovery*) return capture source identifiers for all available capture sources. However, in some situations, it may be desirable to use a preconfigured capture source identifier, for instance, when the frame grabber and parameters are known to the application.

A capture source identifier is formatted in name value pairs separated by the color (':') character. Names are not case sensitive. Consider the following capture source identifier:

```
"fgtype=SAPERA:server=Xcelera-CL_PX4_1:resourceID=0:type=0".
```

This capture source identifier (above) specifies a Sapera frame grabber with a Camera Link interface. The following table documents the possible name values pairs.

Name	Description
fgType	The frame grabber type. Currently, only SAPERA is supported.
type	Specifies the resource type. Possible values are 0 (Camera Link) or 1 (GigE)
serverIndex	Specifies the Sapera server index (i.e. the board number). Applies when fgType=SAPERA and type=0.
server	Specifies the Sapera server name. Required when identifying a GigE capture source. Applies when fgType=SAPERA and type=0|1.
resourceID	Specifies the Sapera resource index. Applies when fgType=SAPERA and type=0|1.

EXAMPLES

Initializing and Deintializing a Capture Source

```
void InitDeinitExample()
{
  uint32_t num_capture_sources = 0;
  int32_t result = IPS_GetCaptureSourceCount( IPS_DEVICE_TYPE_CAMERALINK,
                                              &num_capture_sources);
  if (IPS_FAILED(result))
  {
    cout << "Failed to get capture source due to an error.  Error code = "
         << result << endl;
    return;
  }

  if (num_capture_sources == 0)
  {
    cout << "No capture sources were found." << result << endl;
    return;
  }

  // Get the first capture source
  char capture_source[IPS_MAX_CAPTURE_SOURCE_BYTES];
  char capture_source_descr[IPS_MAX_CAPTURE_SOURCE_BYTES];
  result = IPS_GetCaptureSource(
                      IPS_DEVICE_TYPE_CAMERALINK,
                      0,
                      capture_source, sizeof(capture_source),
                      capture_source_descr, sizeof(capture_source_descr));
  if (IPS_FAILED(result))
  {
    cout << "Failed to get capture source.  Error code = " << result << endl;
    return;
  }

  // Initialize the library with the capture source
  HANDLE_IPS_ACQ handle_ips = NULL;
  result = IPS_InitAcq( CAM_ID_SBF161,  // Configure the SBF 161 camera
                        capture_source,
                        DEFAULT_CONFIG,
                        "$(IPS_SDK_DATA_DIR)\\license.lcx",
                        &handle_ips);

  if (IPS_SUCCEEDED(result))
  {
    // IPS frame acquisition is initialized
    // <<< Do useful things here >>>
    cout << "IPS frame acquisition is initialized.  Capture source is "
         << capture_source << endl;
  }

  if (handle_ips)
  {
    // Always clean up.
    IPS_DeinitAcq(handle_ips);
  }
}
```

```
int32_t AcquireFramesExample(HANDLE_IPS_ACQ handle_ips)
{
  uint32_t frame_width = 128;
  uint32_t frame_height = 128;
  int bytes_per_pixel = 2;
  int frame_data_size = frame_width * frame_height * bytes_per_pixel;

  // Configure the frame acquisition window to acquire 128x128 frames
  int32_t result =  IPS_SetFrameWindow(handle_ips,
                                       0,
                                       0,
                                       frame_width,
                                       frame_height);
  if (IPS_FAILED(result)) return result;

  // Display the camera and frame grabber diagnostic data
  vector<char> diag_buffer(IPS_MAX_DIAGNOSTIC_STRING_BYTES);
  uint32_t diag_buffer_size(0);
  result = IPS_GetCameraDiagnostics( handle_ips,
                                     diag_buffer.data(),
                                     (uint32_t) diag_buffer.size(),
                                     &diag_buffer_size);
  if (IPS_FAILED(result)) return result;
  if (strlen(diag_buffer.data()) == diag_buffer_size-1)
    cout << "Camera diagnostics : " << string(diag_buffer.data()) << endl;
  result = IPS_GetFrameGrabberDiagnostics( handle_ips,
                                           diag_buffer.data(),
                                           (uint32_t) diag_buffer.size(),
                                           &diag_buffer_size);
  if (IPS_FAILED(result)) return result;
  if (strlen(diag_buffer.data()) == diag_buffer_size-1)
    cout << "Frame grabber diagnostics : " << string(diag_buffer.data()) << endl;

  // Start capturing a block of 1000 frames
  VMemory<uint8_t> buffer(frame_data_size*1000);
  result = IPS_StartGrabbing( handle_ips,
                              1000,          // Capture 1000 frames then stop capturing
                              buffer.data(), // User allocated buffer
                              buffer.size(), // size of user allocated buffer
                              false);        // No wrap
  if (IPS_FAILED(result)) return result;

  // Wait for all 1000 frames to be acquired and obtain a pointer to the 1000th frame
  uint64_t frame_number;
  uint8_t * p_frame = NULL;
  result = IPS_WaitFrame(handle_ips,
           1000,                         // Wait until the number of frame has been captured
           IPS_TIMEOUT_WAIT_FOREVER,     // Don't time out, wait for ever
           false,                        // No pause after WaitFrame returns
           &p_frame,                     // Return a pointer to the frame
           &frame_number);               // Return the frame number of the returned frame
  if (IPS_FAILED(result)) return result;
  cout << "Successfully acquired 1000 frames" << endl;

  // Stop acquiring frames
  result = IPS_StopGrabbing(handle_ips);
  return result;
}
```

```
int32_t ContinuousFrameAcquisitionExample(HANDLE_IPS_ACQ handle_ips)
{
  uint32_t frame_width = 128;
  uint32_t frame_height = 128;
  int bytes_per_pixel = 2;
  int frame_data_size = frame_width * frame_height * bytes_per_pixel;

  // Configure the frame acquisition window to acquire 128x128 frames
  int32_t result =  IPS_SetFrameWindow(handle_ips,
                                       0,
                                       0,
                                       frame_width,
                                       frame_height);
  if (IPS_FAILED(result)) return result;

  // Start a continuous capture into a ring buffer capable of
  // storing 1000 frames
  VMemory<uint8_t> buffer(frame_data_size*1000);
  result = IPS_StartContinuousGrabbing(handle_ips,
                                           buffer.data(), // User allocated buffer
                                           buffer.size()); // size of user allocated buffer
  if (IPS_FAILED(result)) return result;

  // Sample 10 frames
  for (int i = 0; i < 10; i++)
  {
    // Wait for the next frame to be acquired and automatically pause
    // frame acquisition to give time to process the frame(s).
    uint64_t frame_number;
    uint8_t * p_frame = NULL;
    result = IPS_WaitFrame(handle_ips,
             IPS_GET_NEXT_FRAME,      // Wait until the number of frame has been captured
             IPS_TIMEOUT_WAIT_FOREVER,// Don't time out, wait for ever
             true,                    // Pause after WaitFrame returns
             &p_frame,                // Return a pointer to the frame
             &frame_number);          // Return the frame number of the returned frame
    if (IPS_FAILED(result)) return result;

    cout << "Acquired frame number : " << frame_number << endl;

    // Resume frame acquistion (it was paused in call to IPS_WaitFrame)
    result = IPS_ResumeGrabbing(handle_ips);
    if (IPS_FAILED(result)) return result;

    // Sleep for a bit to mimick a GUI sampling frames
    Sleep(300);
  }

  // Stop acquiring frames
  result = IPS_StopGrabbing(handle_ips);
  return result;
}
```

```
int32_t JamSyncModeExample(HANDLE_IPS_ACQ handle_ips)
{
  // Set Jam Sync to IPS_JAM_SYNC_RS422_IN and verify
  // by reading it back
  int32_t result = IPS_SetInt32Param( handle_ips,
                                      IPS_WRITE_JAM_SYNC,
                                      IPS_JAM_SYNC_RS422_IN);
  if (IPS_FAILED(result)) return result;

  int jam_sync_mode(IPS_JAM_SYNC_NONE);
  result = IPS_GetInt32Param( handle_ips,
                              IPS_READ_JAM_SYNC,
                              &jam_sync_mode);
  if (IPS_FAILED(result)) return result;
  if (jam_sync_mode == IPS_JAM_SYNC_RS422_IN)
    cout << "Successfully set JAM Synch mode to RS422_IN" << endl;

  // Reset Jam Sync to IPS_JAM_SYNC_NONE and verify by reading
  // it back
  result = IPS_SetInt32Param( handle_ips,
                              IPS_WRITE_JAM_SYNC,
                              IPS_JAM_SYNC_NONE);
  if (IPS_FAILED(result)) return result;

  result = IPS_GetInt32Param( handle_ips,
                              IPS_READ_JAM_SYNC,
                              &jam_sync_mode);
  if (IPS_FAILED(result)) return result;
  if (jam_sync_mode == IPS_JAM_SYNC_NONE)
    cout << "Successfully reset JAM Synch mode to NONE" << endl;

  return result;
}
```

```
int32_t CounterTestModeExample(HANDLE_IPS_ACQ handle_ips, bool enable)
{
  // Set counter test mode enable/disable and verify
  // by reading it back
  int32_t result = IPS_SetInt32Param( handle_ips,
                                      IPS_WRITE_COUNTER_TEST_MODE,
                                      (int32_t) enable);
  if (IPS_FAILED(result)) return result;

  int counter_test_mode_enable(0);
  result = IPS_GetInt32Param( handle_ips,
                              IPS_READ_COUNTER_TEST_MODE,
                              &counter_test_mode_enable);
  if (IPS_FAILED(result)) return result;
  if (counter_test_mode_enable == (int32_t) enable)
    cout << "Successfully set counter test mode to :"
         << counter_test_mode_enable << endl;
  return result;
}
```

```
int32_t IntegrationTimeCalculateExample(HANDLE_IPS_ACQ handle_ips)
{
  // Verify the integration time matches the the calculated integration time
  // using integration row and col ticks
  double integration_time_readout(0);
  int32_t result = IPS_GetDoubleParam(handle_ips,
                                      IPS_READ_INTEGRATION_TIME,
                                      &integration_time_readout);
  if (IPS_FAILED(result)) return result;

  int integration_col_ticks(0);
  int integration_row_tick(0);
  result = IPS_GetInt32Param( handle_ips,
                              IPS_GET_INTEGRATION_COL_TICKS,
                              &integration_col_ticks);
  if (IPS_FAILED(result)) return result;
  result = IPS_GetInt32Param( handle_ips,
                              IPS_GET_INTEGRATION_ROW_TICKS,
                              &integration_row_tick);

  double tm_seconds(0);
  result = IPS_CalculateIntegrationTime(handle_ips,
                                        integration_row_tick,
                                        integration_col_ticks,
                                        &tm_seconds);
  if (IPS_FAILED(result)) return result;
  if (integration_time_readout == tm_seconds)
    cout << "Integration time from camera matches calculated integration time"
         << endl;
  return result;
}
```

```
int32_t IntegrationTimeWriteExample(HANDLE_IPS_ACQ handle_ips,
                                    int row_ticks,
                                    int col_ticks)
{
  // First write the desired row and col integration ticks
  int32_t result = IPS_SetInt32Param( handle_ips,
                                      IPS_WRITE_INTEGRATION_ROW_TICKS,
                                      row_ticks);
  if (IPS_FAILED(result)) return result;

  result = IPS_SetInt32Param( handle_ips,
                              IPS_WRITE_INTEGRATION_COL_TICKS,
                              col_ticks);
  if (IPS_FAILED(result)) return result;

  // Write the integration time using the last stored row and col ticks
  result = IPS_SetInt32Param( handle_ips,
                              IPS_WRITE_INTEGRATION_TIME,
                              0 /*not used*/);
  if (IPS_FAILED(result)) return result;
  cout << "Integration time successfully written to camera" << endl;
  return result;
}
```