TELEDYNE SCIENTIFIC & IMAGING
Everywhereyoulook™

API REFERENCE MANUAL

SBF EMULATION API

TABLE OF CONTENTS

INTRODUCTION

This document describes the SBF Emulation API. This API is a subset of the original SBF API and internally calls the IPS API in order to emulate the SBF API behavior.

SYSTEM OVERVIEW

Figure 1 shows the software architecture, depicting the SBF Emulation Library.

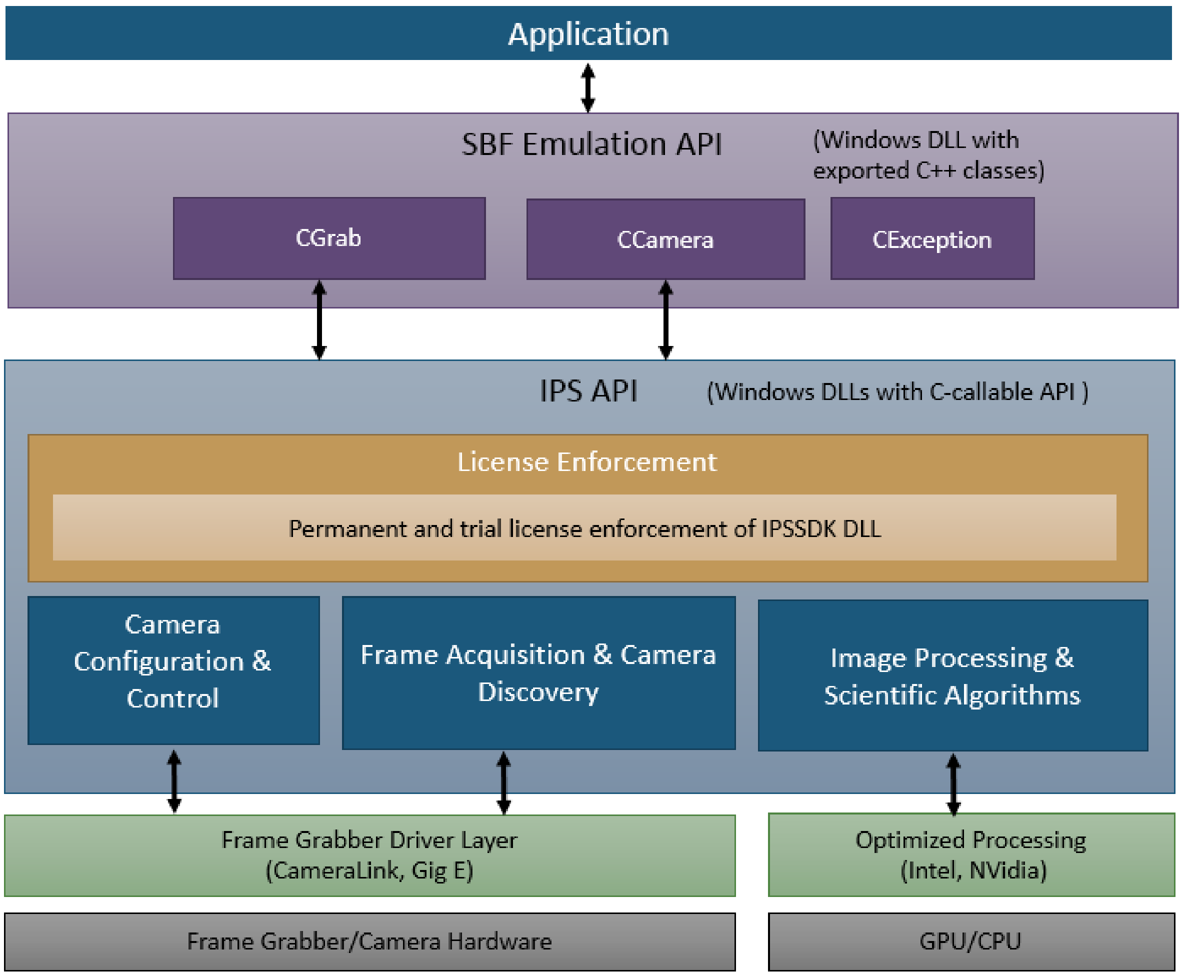
Application
SBF Emulation API
(Windows DLL with exported C++ classes)
CGrab
CCamera
CException
IPS API
(Windows DLLs with C-callable API)
License Enforcement
Permanent and trial license enforcement of IPSSDK DLL
Camera Configuration & Control
Frame Acquisition & Camera Discovery
Image Processing & Scientific Algorithms
Frame Grabber Driver Layer (CameraLink, Gig E)
Optimized Processing (Intel, NVidia)
Frame Grabber/Camera Hardware
GPU/CPU

Figure 1. The SBF Emulation Library provides classes that emulate the original SBF classes: CGrab, CCamera, and CException. Internally, the library utilizes the IPS API to access frame acquisition and camera configuration & control functions.

API CONVENTIONS

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 operation. For example, a pointer to the ring-buffer provided in the CGrab::Start method 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 buffer pointer passed to CGrab::Start must be a virtual pointer 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 *Using CGrab to Acquire Data Frames* for an example on using VMemory to allocate a frame buffer.

Singleton Pattern

The CGrab and CCamera classes utilize the well-known singleton pattern for instantiation. They do not provide public constructors. Instead, they provide a static method called *Instance* that returns a pointer to the one, and only, instance of the class. A static method called *Destroy* is provided that will destroy the single instance of the associated class.

Exceptions

The SBF Emulation classes may throw exceptions using the CException class. No other exception type will be thrown by the library. It is important for calling applications to implement exception handling for those functions that are documented to throw exceptions. The CException::MsgGet function provides a description of the exception. Some of these exceptions may contain error codes associated with the underlying IPS API function call. The CException::StatusCode method will return any IPS API status codes associated with the exception. These status codes are documented in the IPS API Reference documentation.

HEADER FILES, LIBRARIES, AND REDISTRIBUTABLES

The IPS SDK installation package will install the required SBF Emulation 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 SBF Emulation library requires the Visual Studio 2010 SP1 runtime DLLs. The installer will include the Visual Studio runtime installers (both x86 and x64) in the *<Install Dir>\Redistributables* directory.

Linking to the SBF Emulation Library

The SBF Emulation Library is distributed as a Windows DLL called *sbfem.dll.* The import library for this DLL is called *sbfem.lib* and is located under the *<Install Dir>\Lib* directory.

Dependencies

The SBF Emulation Library depends on the following DLLs and libraries: *ipsacq.dll, ipssaperalt.dll, VS 2010 SP1 runtime libraries, SaperaLT 7.5 or newer.*

SYSTEM REQUIREMENTS

The SBF Emulation Library currently supports Windows 7, Windows 8, and Windows 8.1. Both x86 and native x64 libraries are available.

For frame acquisition, the underlying IPS libraries currently requires Dalsa Sapera LT 7.5 or newer. Both GigE Vision and Camera Link are supported by the IPS SDK. The IPS SDK does *NOT* install Dalsa Sapera and the associated frame grabber drivers.

The SBF Emulation Library requires the Visual Studio 2010 SP1 runtime libraries. For convenience, the Visual Studio runtime installers are part of the IPS SDK installation package.

CLASS / TYPE REFERENCE

The SBF Emulation Library provides three class: *CCamera*, *CGrab*, and *CException*. The CCamera class contains methods that can be used to configure and control a directly attached camera using a serial communication link. The CGrab class provides methods for acquiring frame data from the attached camera. Some methods of the CCamera class and CGrab class may throw exceptions. The type of exception will always be CException. The CException class provides a string description of the exception and, where applicable, a status code obtained from an IPS API function call.

CCamera Class

The CCamera class contains methods that can be used to configure and control a directly attached camera using a serial communication link.

Fully Qualified Name
`SBF::CCamera`

Header File
`Camera.h`

Instance

Gets the unique instance of the CCamera class.

Syntax

```
static CCamera* Instance();
```

Return value

A pointer to the unique instance of the CCamera class.

Remarks

CCamera is a singleton class. Only one instance of CCamera can exist at a time. Do not use the delete operator on the pointer returned by *Instance*. Instead, call *Destroy* when you are completely finished using the CCamera object.

Throws

Throws *CException* if an error happens during IPS frame acquisition initialization.

Examples

See *Using CCamera to Configure a Camera*.

See Also

CCamera::Destroy

Destroy

Deletes the unique instance of the CCamera class

Syntax

```
static void Destroy();
```

Remarks

Deletes the singleton object returned by the Instance function and performs additional cleanup. Destroy should only be called once, after you are completely done using the SBF Emulation classes, because the Instance function must perform extensive initialization if it is called again.

Examples

See *Using CCamera to Configure a Camera*.

See Also

CCamera::Instance

FrameRateGet

Gets the current frame rate, in hertz.

Syntax

```
float FrameRateGet() const;
```

Return value

The current frame rate in Hz.

Remarks

This function does not communicate with the camera head.

IntegrationTimeGet

Gets the current integration time, in seconds.

Syntax

```
float IntegrationTimeGet() const;
```

Return value

The current integration time in seconds.

Remarks

Uses the number of integration and row ticks previously read from the camera to calculate the integration time, in seconds. This function does not communicate with the camera head.

See Also

CCamera::IntegrationTimeWrite

IntegrationTimeWrite

Write an integration time to the camera head.

Syntax

```
bool IntegrationTimeWrite(
  int iRowTicks,
  int iColTicks,
  int iSuperframeIndex = 0
);
```

Parameters

iRowTicks [in]
The desired integration row ticks.

iColTicks [in]
The desired integration column ticks.

iSuperframeIndex [in]
This value is ignored. The SBF Emulation Library, and IPS API, do not support super frames.

Return value

true if the write succeeded, otherwise false.

Remarks

The integration time is determined by the number of integration row and column ticks stored on the camera head. This function allows you to set the integration time by choosing the number of integration row and column ticks directly.

See Also

CCamera::IntegrationTimeGet

IntegrationTimeCalculate

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

Syntax

```
float IntegrationTimeCalculate(
  int iRowTicks,
  int iColTicks
);
```

Parameters

iRowTicks [in]
The integration row ticks.

iColTicks [in]
The integration column ticks.

Return value

The integration time, in seconds.

IntegrateWhileReadModeWrite

Changes the camera head integration mode to Integrate While Read.

Syntax

```
bool IntegrateWhileReadModeWrite();
```

Return value

true if the write succeeded, otherwise false.

Remarks

The integration time mode for most SBF cameras should be in *Integrate While Read* (*Fast Frame Transfer*) mode, rather than in *Integrate Then Read* mode. This function has no effect on FPAs which don't support *Integrate While Read* mode.

JamSyncWrite

Sets the jam sync state of the camera head.

Syntax

```
bool JamSyncWrite(JamSyncType jam_sync_type);
```

Parameters

jam_sync_type [in]
The desired synchronization state of the camera. See *JamSyncType Enumeration*.

Return value

true if the write succeeded, otherwise false.

Remarks

The jam sync state determines how the camera is synchronized with external signals.

FpaColumnStartGet

Gets the zero-based index of the first column on the focal plane which will transmit data.

Syntax

```
int FpaColumnStartGet() const;
```

Return value

The first column of the focal plane window.

Remarks

This function does not communicate with the camera head.

FpaColumnEndGet

Gets the zero-based index of the last column on the focal plane which will transmit data.

Syntax

```
int FpaColumnEndGet() const;
```

Return value

The last column of the focal plane window.

Remarks

This function does not communicate with the camera head.

FpaRowStartGet

Gets the zero-based index of the first row on the focal plane which will transmit data.

Syntax

```
int FpaRowStartGet() const;
```

Return value

The first row of the focal plane window.

Remarks

This function does not communicate with the camera head.

FpaRowEndGet

Gets the zero-based index of the last row on the focal plane which will transmit data.

Syntax

```
int FpaRowEndGet() const;
```

Return value

The last row of the focal plane window.

Remarks

This function does not communicate with the camera head.

NumColTicksGet

Gets the number of timing column ticks needed to read one row of pixel data.

Syntax

```
int NumColTicksGet() const;
```

Return value

The number of timing ticks to read one row of pixel data.

Remarks

This number is related to the number of FPA rows and can be useful for diagnostics. This function does not communicate with the camera head.

NumRowTicksGet

Gets the number of timing row ticks needed to read one frame of pixel data.

Syntax

```
int NumRowTicksGet() const;
```

Return value

The number of timing ticks to read one frame of pixel data.

Remarks

This number is related to the number of FPA rows and can be useful for diagnostics. This function does not communicate with the camera head.

Diagnostics

Returns current camera configuration and state formatted as an ASCII string.

Syntax

```
const char * Diagnostics();
```

Return value

A string containing camera diagnostic information.

FpaTypeGet

Returns the FPA type of the current camera.

Syntax

```
FpaType FpaTypeGet() const;
```

Return value

The *FpaType* of the current camera.

Remarks

This function does not communicate with the camera head.

SerialNumGet

Gets the serial number stored in the camera head.

Syntax

```
int SerialNumGet() const;
```

Return value

The serial number stored in the camera head.

Remarks

The serial number returned should agree with the 6-digit number marked on the dewar (e.g. 00-4398 would return 4398, while 99-0345 would return 990345). This function does not communicate with the camera head.

FrameCounterEnabledWrite

Sets the FrameCounterEnabled bit on the camera head.

Syntax

```
bool FrameCounterEnabledWrite(bool bFrameCounter);
```

Parameters

bFrameCounter [in]

Set to true to enable the frame counter, false to disable it.

Return value

true if the write succeeded, otherwise false.

Remarks

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.

TemperatureFpaRead

Reads the FPA temperature from the camera head.

Syntax

```
float TemperatureFpaRead() const;
```

Return value

The temperature of the FPA, in Kelvin.

CounterTestModeWrite

Turns the electronics test mode on or off.

Syntax

```
bool CounterTestModeWrite(bool bTestMode);
```

Parameters

bTestMode [in]

Set to true to turn test mode on, false to turn it off.

Return value

Set to true to turn test mode on, false to turn it off.

Remarks

The test mode outputs a counter that ramps repeatedly from 0 to 16383. This is done without the help of the focal plane, so it can be used to test the operation of the camera electronics in isolation from the focal plane.

NumOutputsGet

Gets the number of A/D converters used by the current focal plane.

Syntax

```
int NumOutputsGet() const;
```

Return value

Returns the number of A/D converters used by the current focal plane.

Remarks

Usually returns 1, 2, 4, or 16. This function does not communicate with the camera head.

FpaDetectorBiasGet

Gets the FPA detector bias value.

Syntax

```
int FpaDetectorBiasGet() const;
```

Return value

The detector bias value, from 0 to 255 or -1 if bias not available.

Remarks

Some FPAs or electronics may support only 8, 16, or 32 different values. The FPA detector bias is pre-set to give optimum performance and it should not generally need to be changed for InSb detectors.

This function does not communicate with the camera head.

FpaDetectorBiasWrite

Writes the FPS detector bias to the camera head.

Syntax

```
int FpaDetectorBiasWrite(int iValue);
```

Parameters

iValue [in]

The desired value of the FPA detector bias. Must be between 0 and 255, inclusive. Otherwise, it is truncated to fit within this range.

Return value

The FPA detector bias value

Remarks

It is recommend that you not use this function. The FPA detector bias is pre-set to give optimum performance and it should not generally need to be changed for InSb detectors.

This function does not communicate with the camera head.

handle_ips

Returns the underlying IPS API handle.

Syntax

```
void * handle_ips();
```

Return value

The IPS API handle

The CGrab class provides methods for acquiring frame data from an attached camera/sensor.

Fully Qualified Name

`SBF::CGrab`

Header File

`Grab.h`

Instance

Gets the unique instance of the CGrab class.

Syntax

```
static CGrab* Instance();
```

Return value

A pointer to the unique instance of the CGrab class.

Remarks

CGrab is a singleton class. Only one instance of CGrab can exist at a time. Do not use the delete operator on the pointer returned by *Instance*. Instead, call *Destroy* when you are completely finished using the CGrab object.

Throws

Throws *CException* if an error happens during IPS library initialization.

Examples

See *Using CGrab to Acquire Data Frames*

See Also

CCamera::Destroy

Destroy

Deletes the unique instance of the CGrab class

Syntax

```
static void Destroy();
```

Remarks

Deletes the singleton object returned by the Instance function and performs additional cleanup. Destroy should only be called once, after you are completely done using the SBF Emulation classes, because the Instance function must perform extensive initialization if it is called again.

Examples

See *Using CGrab to Acquire Data Frames*

See Also

CGrab::Instance

Start

Starts frame acquisition into a memory buffer.

Syntax

```
bool Start(
  unsigned short * p_img_buffer,
  unsigned long buffer_length,
  unsigned int iFramesToGrab = 0
);
```

Parameters

p_img_buffer [in]
A pointer to a buffer that has been allocated by the caller. This pointer is expected to be a Windows virtual pointer and must be allocated using VirtualAlloc (see remarks below).

buffer_length [in]
The number of unsigned short elements allocated in *p_img_buffer*.

iFramesToGrab [in]

The number of frames to grab. Set iFramesToGrab to 0 for an infinite grab. In this case, RAM is filled in ring-buffer fashion.

Return value

Returns true if successfully started frame acquisition, otherwise false.

Remarks

This function returns immediately after starting the frame acquisition thread. Use *Wait*() to wait for a given frame. Call *Stop*() to stop a grab and free up resources. The buffer pointed to by p_img_buffer can be smaller than is necessary to hold *iFramesToGrab* frames of data. In this case, the *p_img_buffer* buffer will be filled in ring fashion. *p_img_buffer* should be large enough that you can process the data before it is overwritten by fresh data.

The buffer pointed to by p_img_buffer must be allocated using VirtualAlloc. A helper class called VMemory is provided to simplify management of a virtual buffer. See *Using CGrab to Acquire Data Frames* for an example on using VMemory to allocate the buffer passed to this method.

Every call to Start() must be paired with a call to Stop().

Throws

Throws *CException* if an error occurs when starting frame acquisition.

Examples

See *Using CGrab to Acquire Data Frames*

See Also

CGrab::Stop

Wait

Wait for the specified frame and, if successful, returns a pointer to the frame data.

Syntax

```
int Wait(
  unsigned short** ppUShort,
  int nFrameToWaitFor = -1,
  int nMilliSecWait = -1
);
```

Parameters

ppUShort [in]

The address of a pointer. It may be 0. If it is nonzero, then when the function returns it will hold a pointer to a frame. This may not be the requested frame, but it will correspond to the return value. This pointer will be point somewhere within the buffer passed to Start().

nFrameToWaitFor [in]

The number of the frame to wait for. Set *nFrameToWaitFor* to -1 to wait for the next available frame.

nMilliSecWait [in]

The time in milliseconds to wait before returning if the frame hasn't been grabbed. Set *nMilliSecWait* to -1 for an infinite wait.

Return value

Returns the frame number if successful. Otherwise, -1 is returned.

Remarks

This function is to be used during a grab that was started using Start(). It returns when the frame is grabbed or when the timeout period has been exceeded.

If the requested frame is already in memory when this function is called, the returned frame number will be that of the most recently grabbed frame, which may not be the requested frame. If the requested frame is not in memory, it will wait for the requested frame and return its frame number. If it cannot wait for the requested frame, either because it times out or because the frame has been overwritten, it returns -1.

Throws

Throws *CException* if an error occurs when waiting for a frame.

Examples

See *Using CGrab to Acquire Data Frames*

See Also

CGrab::Start

Stop

Stop grabbing images and return a pointer to the newest image

Syntax

```
int Stop(
  unsigned short** ppUShort,
  int nFrameToWaitFor = -1,
  int nMilliSecWait = -1
);
```

Parameters

ppUShort [in]

The address of a pointer. It may be 0. If it is nonzero, then when the function returns it will hold a pointer to a frame. This may not be the requested frame, but it will correspond to the return value. This pointer will be point somewhere within the buffer passed to Start().

nFrameToWaitFor [in]

The number of the frame to wait for. Set *nFrameToWaitFor* to -1 to wait for the next available frame.

nMilliSecWait [in]

The time in milliseconds to wait before returning if the frame hasn't been grabbed. Set *nMilliSecWait* to -1 for an infinite wait.

Return value

Returns the frame number if successful. Otherwise, -1 is returned.

Remarks

This function is to be used to end a grab that was started using Start(). It works the same as Wait() except that it stops the grab and frees up resources after it waits for the given frame. It returns when the frame is grabbed or when the timeout period has been exceeded.

If the requested frame is already in memory when this function is called, the returned frame number will be that of the most recently grabbed frame, which may not be the requested frame. If the requested frame is not in memory, it will wait for the requested frame and return its frame number. If it cannot wait for the requested frame, either because it times out or because the frame has been overwritten, it returns -1.

Throws

Throws *CException* if an error occurs while waiting for a frame or when stopping frame acquisition.

Examples

See *Using CGrab to Acquire Data Frames*

See Also

CGrab::Start

GrabSizeGet

Returns the frame width and height currently being grabbed.

Syntax

```
std::pair<int, int> GrabSizeGet() const;
```

Return value

Returns the width and height of the frame data being grabbed. The width and height tuple are returned in a *std::pair<width, height>* object.

Diagnostics

Returns current frame grabber configuration and state formatted as an ASCII string.

Syntax

```
const char * Diagnostics();
```

Return value

Returns a string containing frame grabber diagnostic information.

handle_ips

Returns the underlying IPS API handle.

Syntax

```
void * handle_ips();
```

Return value

The IPS API handle

CException Class

Some methods of the CCamera class and CGrab class may throw exceptions. The type of exception will always be CException. The CException class provides a string description of the exception and, where applicable, a status code obtained from an IPS API function call

Fully Qualified Name
`SBF::CException`

Header File
`Exception.h`

CException

CException constructor.

Syntax

```
CException();

  OR

CException(
  const char* szMsg,
  int32_t status_code = 0
);
```

Parameters

szMsg [in]
A to a string that describes the exception.

status_code [in]
IPS API status code representing the low level IPS API error code that resulted in the exception.

~CException

CException destructor.

Syntax

```
virtual ~CException();
```

MsgGet

Returns a null terminated ASCII string containing a description of the exception.

Syntax

```
const char * MsgGet();
```

StatusCode

Returns the IPS API error code that resulted in the exception.

Syntax

```
int32_t StatusCode();
```

See Also

See the IPS API Reference documentation for a listing of the possible IPS API error codes.

JamSyncType Enumeration

The JamSyncType is used to set the jam sync mode of the camera, i.e. how it synchronizes with external signals.

```
enum JamSyncType
{
  // The camera will be free-running.
  NONE,

  // The camera will be free-running,
  // but it will output a TTL pulse at the start of every frame.
  TTL_OUT,

  // The camera is in jam sync mode.  It will grab a frame each
  // time it receives a TTL pulse and will not grab any frames otherwise.
  TTL_IN,

  // The camera is in jam sync mode. It will grab a frame each time it
  // receives a signal through the RS422 port and will not grab any frames otherwise.
  RS422_IN
};
```

FpaType Enumeration

The FpaType enumeration is used to identify the current attached camera/sensor. Currently, only the FpaType::FPA_SBF161 sensor is supported by the SBF Emulation Library.

```
enum FpaType
{
  NOT_SPECIFIED = 0, /// FPA not specified
  FPA_SBFJ108 = 1,  /// SBFJ108 64x64 LWIR HgCdTe
  FPA_SBF119 = 2, /// SBF119 640x512 InSb Windowing
  FPA_SBF125 = 3, /// SBF125 320x256 InSb Windowing
  FPA_SBF134 = 4, /// SBF134 256x256 InSb SharpIntegration, 3Windows
  FPA_SBF135 = 5, /// SBF135 256x256 InSb RollingModeIntegration, 1Output
  FPA_SBF136 = 6, /// SBF136 128x128 InSb 32 Output HiSpeed Windowing
  FPA_SBF141 = 7, /// SBF141 640x512 InSb Windowing Quiet
  FPA_SBF161 = 8, /// SBF161 128x128 LWIR HgCdTe Windowing
  FPA_EGL2333  = 9, /// EGL2333 256x256 LWIR HgCdTe
  FPA_SBF162 = 10, /// SBF162 640x512 InSb VoltageMode LowPower Windowing
  FPA_SBF151 = 11, /// SBF151 320x256 InSb VoltageMode Windowing
  FPA_SBF167 = 12, /// SBF167 320x256 InSb VoltageMode Windowing Quiet
  FPA_SBF168 = 13, /// SBF168 320x256 InSb Digital FPA
  FPA_DRS640 = 14, /// DRS640 640x480 HgCdTe LWIR FPA
  FPA_SBF177 = 15, /// SBF177 640x512 InSb Windowing
  FPA_SBF180 = 16, /// SBF180 320x256 30µm VoltageMode LowPower Windowing
  FPA_SBF178 = 17, /// SBF178 640x512 20µm WindowingLowPower
  FPA_SBF184 = 18, /// SBF184 1024x1024 19.5µm Windowing
  FPA_SBF113 = 19, /// SBF113 128x128 InSb
  FPA_SBF191 = 20, /// SBF191 640x512 14-bit Digital, 20µm Windowing
  FPA_SBF193 = 21, /// SBF193 640x512 24µm Windowing
  FPA_SBF194 = 22, /// SBF194 320x256 30µm Windowing
  FPA_SBF196 = 23, /// SBF196 1024x1024 14-bit Digital, 25µm Windowing
  FPA_SBF191DASH = 24, /// SBF191 640x512 14-bit Digital, 20µm Windowing
  FPA_SBF200 = 25, /// SBF200 320x256 14-bit Digital, 30 µm Windowing
  FPA_SBF199 = 26, /// SBF199 640x512 14-bit Digital, 24 µm Windowing
  FPA_SBF204 = 27, /// SBF204 1280x1024 14-bit Digital, 12 µm CTIA, Windowing
  FPA_SBF207  = 28, /// SBF207 1280x1024 14-bit Digital, 12 µm Windowing
  FPA_SBF208 = 29, /// SBF208 1024x1024 14-bit Digital, 25 µm Windowing
  FPA_SBF209 = 30 /// SBF209 640x512 14-bit Digital, 20 µm Windowing
};
```

EXAMPLES

Using CCamera to Configure a Camera

```
int32_t JamSyncExample()
{
  CCamera * p_camera = CCamera::Instance();

  // Set JamSyncWrite to RS422_IN
  int jam_sync_mode(0);
  p_camera->JamSyncWrite(CCamera::RS422_IN);

  // Use the IPS API to read it back
  int32_t result = IPS_GetInt32Param(
                              p_camera->handle_ips(),
                              IPS_READ_JAM_SYNC,
                              &jam_sync_mode);
  if (IPS_FAILED(result)) return result;

  if (jam_sync_mode == CCamera::RS422_IN)
    cout << "Successfully changed jam sync to RS422_IN" << endl;

  // Set Jam sync to None and verify
  p_camera->JamSyncWrite(CCamera::NONE);

  // Use the IPS API to read it back
  result = IPS_GetInt32Param( p_camera->handle_ips(),
                            IPS_READ_JAM_SYNC,
                            &jam_sync_mode);
  if (IPS_FAILED(result)) return result;

  if (jam_sync_mode == CCamera::NONE)
    cout << "Successfully changed jam sync to NONE" << endl;

  return result;
}
```

```
int32_t CounterTestModeExample()
{
  CCamera * p_camera = CCamera::Instance();

  // Turn off frame counter mode initially
  p_camera->CounterTestModeWrite(false);

  // Turn it on and verify that  counter test
  // mode enabled is on by reading from the camera head
  p_camera->CounterTestModeWrite(true);
  int counter_test_mode(0);
  int result = IPS_GetInt32Param( p_camera->handle_ips(),
                                IPS_READ_COUNTER_TEST_MODE,
                                &counter_test_mode);
  if (IPS_FAILED(result)) return result;
  if (counter_test_mode != 0)
    cout << "Successfully enabled counter test mode" << endl;

  // Reset to off and verify that it is off
  p_camera->CounterTestModeWrite(false);
  result = IPS_GetInt32Param( p_camera->handle_ips(),
                            IPS_READ_COUNTER_TEST_MODE,
                            &counter_test_mode);
  if (IPS_FAILED(result)) return result;
  if (counter_test_mode == 0)
    cout << "Successfully disabled counter test mode" << endl;

  return result;
}
```

Using CGrab to Acquire Data Frames

```
bool AcquireFramesExample()
{
  CCamera * p_camera = CCamera::Instance();
  CGrab * p_grab = CGrab::Instance();

  string str_diagnostics = p_camera->Diagnostics();
  cout << "Camera diagnostics = " << str_diagnostics << endl;

  str_diagnostics = p_grab->Diagnostics();
  cout << "Frame grabber diagnostics = " << str_diagnostics << endl;

  const int NUM_FRAMES = 1000;
  uint32_t frame_width = 128;
  uint32_t frame_height = 128;
  VMemory<uint16_t> buffer(frame_width*frame_height*1000);

  // Capture a block of frames then automatically stop
  bool bResult = p_grab->Start( buffer.data(),
                                (unsigned long) buffer.size(),
                                NUM_FRAMES);
  if (!bResult) return bResult;

  uint16_t * p_frame = NULL;
  int frame_number = p_grab->Wait(&p_frame, NUM_FRAMES-1);
  if (frame_number >= 0)
  {
    cout << "Received frame_number = " << frame_number << endl;
    return true;
  }
  else
  {
    return false;
  }
}
```

```
bool ContinuousFrameAcquistionExample()
{
  const int NUM_FRAMES = 1000;
  CCamera * p_camera = CCamera::Instance();
  CGrab * p_grab = CGrab::Instance();
  uint16_t * p_frame = NULL;
  uint32_t frame_width = 128;
  uint32_t frame_height = 128;
  VMemory<uint16_t> buffer(frame_width*frame_height*10);

  // Continuously capture a block of frames, passing a ring buffer
  bool bResult = p_grab->Start( buffer.data(),
                                (unsigned long) buffer.size());
  if (!bResult) return bResult;

  for (int i = 0; i < 10; i++)
  {
    // Get the next available frame
    int frame_number = p_grab->Wait(&p_frame);
    if (frame_number >= 0)
    {
      cout << "Acquired frame_number = " << frame_number << endl;
    }
    else
    {
      return false;
    }
  }

  // Stop grabbing
  p_grab->Stop(&p_frame);

  return true;
}
```