SIDX

Version 7

Bruxton Corporation

Table of Contents

1. Sample

1.1 Acquisition

The acquired images are written to disk and saved in a file.

AcquireByCount

Write a desired number of images to disk during acquisition.

```
public class AcquireByCount {
   public static void main(String[] args) {
      try {
         System.out.println("start");
         test();
         System.out.println("done");
      }
      catch (java.io.IOException exception) {
         System.out.println(exception.getMessage());
      }
   }

   private static void test() throws java.io.IOException {
      com.bruxton.sidx.SIDXRootFactory root_create = new com.bruxton.sidx.SIDXRootFactory();
      String license = "";
      com.bruxton.sidx.SIDXRoot root = root_create.Open(license);

      root.CameraScan();
      // The first connected camera has the index '0'
      String camera_name = root.CameraScanGetName(0);
      System.out.println("open camera (CameraOpenName) " + camera_name);
      com.bruxton.sidx.SIDXCamera camera = root.CameraOpenName(camera_name);

      try {
       cameraSetUp(camera);
      }
      catch (java.io.IOException exception) {
         System.out.println(exception.getMessage());
      }
      camera.Close();
      root.Close();
   }

   final static long limit_count = 1;

   private static void cameraSetUp(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
      System.out.println("set image limit (AcquireImageSetLimit) " + String.valueOf(limit_count));
      camera.AcquireImageSetLimit(limit_count);
      com.bruxton.sidx.SIDXAcquire acquire = camera.AcquireOpen();

      try {
         acquireLimitCount(acquire);
```

```
        }
        catch (java.io.IOException exception) {
            System.out.println(exception.getMessage());
        }
        acquire.Close();
    }

    final static String archive_name = "ImageArchive.tif";

    private static void acquireLimitCount(com.bruxton.sidx.SIDXAcquire acquire) throws java.io.IOException {
        System.out.println("create file (ArchiveOpenNew) " + archive_name);
        acquire.ArchiveOpenNew(archive_name, "TIFF", true);

        System.out.println("start acquisition (Start)");
        acquire.Start();

        long image_count = 0;
        long image_index = 0;
        boolean acquiring = true;
        while (acquiring) {
         acquiring = acquire.GetStatus();
            long current_image = acquire.ImageGetCount();
            long new_image_count = current_image - image_count;
            if (new_image_count > 0) {
                acquire.ArchiveWrite(image_index, new_image_count);
                image_count = current_image;
                image_index = current_image;
            }
        }
        acquire.Stop();
        System.out.println("stop acquisition (Stop), images acquired " + image_count);
    }
}
```

AcquireByTime

Write images to disk during a desired time duration.

```
public class AcquireByTime {
    public static void main(String[] args) {
        try {
            System.out.println("start");
            test();
            System.out.println("done");
        }
        catch (java.io.IOException exception) {
            System.out.println(exception.getMessage());
        }
    }

    private static void test() throws java.io.IOException {
        com.bruxton.sidx.SIDXRootFactory root_create = new com.bruxton.sidx.SIDXRootFactory();
        String license = "";
        com.bruxton.sidx.SIDXRoot root = root_create.Open(license);

        root.CameraScan();
        // The first connected camera has the index '0'.
        String camera_name = root.CameraScanGetName(0);
```

```
    System.out.println("open camera (CameraOpenName) " + camera_name);
    com.bruxton.sidx.SIDXCamera camera = root.CameraOpenName(camera_name);

    try {
        cameraSetUp(camera);
    }
    catch (java.io.IOException exception) {
        System.out.println(exception.getMessage());
    }
    camera.Close();
    root.Close();
}

private static void cameraSetUp(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
    // Image limit zero continuously acquires until Stop or Abort is called.
    camera.AcquireImageSetLimit(0);
    com.bruxton.sidx.SIDXAcquire acquire = camera.AcquireOpen();

    try {
        acquireLimitTime(acquire);
    }
    catch (java.io.IOException exception) {
        System.out.println(exception.getMessage());
    }
    acquire.Close();
}

final static long limit_time = 1000; // milliseconds
final static String archive_name = "ImageArchive.tif";

private static void acquireLimitTime(com.bruxton.sidx.SIDXAcquire acquire) throws java.io.IOException {
    double limit_time_seconds = (double)limit_time / 1000;
    System.out.println("set time limit " + String.valueOf(limit_time_seconds) + " seconds");

    System.out.println("create file (ArchiveOpenNew) " + archive_name);
    acquire.ArchiveOpenNew(archive_name, "TIFF", true);

    System.out.println("start acquisition (Start)");
    acquire.Start();

    long image_count = 0;
    long image_index = 0;

    long start_time = System.currentTimeMillis();
    long end_time = System.currentTimeMillis();

    while (end_time - start_time < limit_time) {
        boolean acquiring = acquire.GetStatus();
        if (!acquiring)
            break;
        long current_image = acquire.ImageGetCount();
        long new_image_count = current_image - image_count;
        if (new_image_count > image_count) {
            acquire.ArchiveWrite(image_index, new_image_count);
            image_count = current_image;
            image_index = current_image;
        }
        end_time = System.currentTimeMillis();
    }
    acquire.Stop();
    double acquire_time = (end_time - start_time) / 1000;
    System.out.println("stop acquisition (Stop), images acquired " + image_count + " in " + acquire_time + " seconds");
}
```

```
}
```

AcquireByTrigger

Acquire images by using a trigger signal.

```
public class AcquireByTrigger {
    public static void main (String[] args) {
        try {
            System.out.println("start");
            com.bruxton.sidx.SIDXRootFactory root_create = new com.bruxton.sidx.SIDXRootFactory();
         String license = "";
            com.bruxton.sidx.SIDXRoot root = root_create.Open(license);

            test(root);

            root.Close();
            System.out.println("done");
        }
        catch (java.io.IOException exception) {
            System.out.println(exception.getMessage());
        }

    }

    private static void test(com.bruxton.sidx.SIDXRoot root){
        try {
            root.CameraScan();
            // The first connected camera has the index '0'.
            String camera_name = root.CameraScanGetName(0);
            System.out.println("open camera (CameraOpenName) " + camera_name);
            com.bruxton.sidx.SIDXCamera camera = root.CameraOpenName(camera_name);

            try {
                cameraSetUp(camera);
            }
            catch (java.io.IOException exception) {
                System.out.println(exception.getMessage());
            }
            camera.Close();
        }
        catch (java.io.IOException exception) {
            System.out.println(exception.getMessage());
        }
    }

    final static com.bruxton.sidx.SIDXTriggerInMode trigger_mode =
        com.bruxton.sidx.SIDXTriggerInMode.ALWAYS;

    final static com.bruxton.sidx.SIDXSignalActiveMode signal_mode =
        com.bruxton.sidx.SIDXSignalActiveMode.EDGE_ANY;

    private static void cameraSetUp(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
        int image_limit = 0;
        camera.AcquireImageSetLimit(image_limit);

        // Set the time interval needed to perform an operation between triggers.
        double set_external_delay = 0.03;
```

```
    camera.ExternalDelaySet(set_external_delay);

    if (camera.TriggerSignalExists(signal_mode))
        camera.TriggerSignalSet(signal_mode);

    switch (trigger_mode) {
    case ALWAYS:
        camera.TriggerModeSet(trigger_mode);
        break;
    case EXPOSURE_START:
        triggerByExposureStart(camera);
        break;
    case EXPOSURE_DURATION:
        triggerByExposureDuration(camera);
        break;
    case SEQUENCE_START:
        break;
    default:
        System.out.println("Unknown trigger mode");
        break;
    }
    System.out.println("set trigger mode (TriggerModeSet) " + trigger_mode);

    com.bruxton.sidx.SIDXAcquire acquire = camera.AcquireOpen();

    try {
        acquireTriggerLimitTime(acquire);
    }
    catch (java.io.IOException exception) {
        System.out.println(exception.getMessage());
    }

    acquire.Close();
}

final static String archive_file_name = "Image.tif";

private static void acquireTriggerLimitTime(com.bruxton.sidx.SIDXAcquire acquire) throws java.io.IOException {
    System.out.println("create file (ArchiveOpenNew) " + archive_file_name);
    acquire.ArchiveOpenNew(archive_file_name, "TIFF", true);

    // Limit the acquisition loop. If using a triggering device, the triggering
    // device should control the acquisition loop duration.
    long limit_time = 800;
    double limit_time_seconds = (double)limit_time / 1000;
    System.out.println("set time limit " + String.valueOf(limit_time_seconds) + " seconds");

    // The value returned by GetImageInterval is the minimum amount of time between triggers.
    // This value takes into account the exposure value and the external delay.
    double trigger_duration = acquire.GetImageInterval();

    // The value returned by GetGapInterval is the interval of time
    // available to perform an peration between exposures.
    double wavelength_switch_duration = acquire.GetGapInterval();

    System.out.println("start acquisition (Start)");
    acquire.Start();

    // Trigger Preparation
    //
    // SIDX_TRIGGER_IN_MODE_EXPOSURE_START
    // If SIDXTriggerInMode.EXPOSURE_START is set, the interval between
    // trigger starts is the value returned by acquire.GetImageInterval.
    // The interval for switching wavelengths is equivalent to the value
```

```
    // set by camera.ExternalDelay.
    //
    // SIDX_TRIGGER_IN_MODE_EXPOSURE_DURATION
    // If SIDXTriggerInMode.EXPOSURE_DURATION is set, the exposure duration
    // is set by the external trigger. The maximum interval between the end
    // of one exposure and the start of the next exposure is the value returned
    // by acquire.GetGapInterval. This interval may be longer than the value
    // set by camera.ExternalDelaySet

    long image_count = 0;
    long image_index = 0;

    long start_time = System.currentTimeMillis();
    long end_time = System.currentTimeMillis();

    boolean acquiring;
    do {
        acquiring = acquire.GetStatus();

        try {
            // During sleep poll for trigger status.
            java.lang.Thread.sleep(10);
        } catch (InterruptedException exception) {
            System.out.println("InterruptedException" + exception.getMessage());
        }

        long current_image_count = acquire.ImageGetCount();
        long new_image_count = current_image_count - image_count;

        if (new_image_count > image_count)
        {
            acquire.ArchiveWrite(image_index, new_image_count);
            image_count = current_image_count;
            image_index = current_image_count;
        }

        if (!acquiring)
            break;
        end_time = System.currentTimeMillis();
    } while (end_time - start_time < limit_time);

    acquire.Abort();
    System.out.println("abort acquisition (Abort), images acquired " + image_count);
}

private static void triggerByExposureStart(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
    if (trigger_mode == com.bruxton.sidx.SIDXTriggerInMode.EXPOSURE_START)
    {
        if (camera.TriggerModeExists(trigger_mode))
        camera.TriggerModeSet(trigger_mode);
    }

    double set_exposure_duration = 0.01;
    System.out.println("set exposure (ExposeSet) " + set_exposure_duration);
    camera.ExposeSet(set_exposure_duration);

    double exposure_value = camera.ExposeGetValue();
    if (exposure_value != set_exposure_duration)
        System.out.println("actual exposure value (ExposeGetValue) " + exposure_value);
}

private static void triggerByExposureDuration(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
    if (trigger_mode == com.bruxton.sidx.SIDXTriggerInMode.EXPOSURE_DURATION)
    {
```

```
            if (camera.TriggerModeExists(trigger_mode))
            camera.TriggerModeSet(trigger_mode);
        }
    }
}
```

SetParametersAndArchive

To check if a specific kind of parameter can be set for the camera, for example intensifier gain, first check the setting type, if it returns NONE the parameter is not supported by the camera.

```
public class SetParametersAndArchive {
    public static void main (String[] args) {
        try {
            System.out.println("start");
            test();
            System.out.println("done");
        }
        catch (java.io.IOException exception) {
            System.out.println(exception.getMessage());
        }
    }

    private static void test() throws java.io.IOException {
        com.bruxton.sidx.SIDXRootFactory root_create = new com.bruxton.sidx.SIDXRootFactory();
        String license = "";
        com.bruxton.sidx.SIDXRoot root = root_create.Open(license);

        root.CameraScan();
        // The first connected camera has the index '0'.
        String camera_name = root.CameraScanGetName(0);
        System.out.println("open camera (CameraOpenName) " + camera_name);
        com.bruxton.sidx.SIDXCamera camera = root.CameraOpenName(camera_name);

        try {
            cameraSetParameters(camera);
        }
        catch (java.io.IOException exception) {
            System.out.println(exception.getMessage());
        }
        camera.Close();
        root.Close();
    }

    final static int set_binning_x = 1;
    final static int set_binning_y = 1;
    final static double set_exposure_duration = 0.01;
    final static double set_analog_gain = 1.0;
    final static int set_em_gain = 1;
    final static double set_intensifier_gain = 1.0;
    final static long set_image_limit = 1;

    final static com.bruxton.sidx.SIDXSettingType setting_type_none = com.bruxton.sidx.SIDXSettingType.NONE;
    final static com.bruxton.sidx.SIDXTriggerInMode trigger_mode = com.bruxton.sidx.SIDXTriggerInMode.ALWAYS;

    private static void cameraSetParameters(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
        com.bruxton.sidx.SIDXBinning set_binning = new com.bruxton.sidx.SIDXBinning(set_binning_x, set_binning_y);
        System.out.println("set binning (BinningSet) x " + set_binning_x + ", y " + set_binning_y);
```

```
    camera.BinningSet(set_binning);

    System.out.println("set exposure (ExposeSet) " + set_exposure_duration + " seconds");
    camera.ExposeSet(set_exposure_duration);

    // A setting type of NONE indicates the camera does not support analog gain.
    if (camera.GainGetType() != setting_type_none) {
        System.out.println("set gain (GainSet) " + set_analog_gain);
        camera.GainSet(set_analog_gain);
    }
    // A setting type of NONE indicates the camera does not support EM gain.
    if (camera.EMGainGetType() != setting_type_none) {
        System.out.println("set em gain (EMGainSet) " + set_em_gain);
        camera.EMGainSet(set_em_gain);
    }
    // A setting type of NONE indicates the camera does not support intensifier gain.
    if (camera.IntensifierGetType() != setting_type_none) {
        System.out.println("set intensifier gain (IntensifierSet) " + set_intensifier_gain);
        camera.IntensifierSet(set_intensifier_gain);
    }

    if (camera.TriggerModeExists(trigger_mode)) {
        System.out.println("set trigger input (TriggerModeSet) " + trigger_mode);
        camera.TriggerModeSet(trigger_mode);
    }

    System.out.println("set image limit (AcquireImageSetLimit) " + set_image_limit);
    camera.AcquireImageSetLimit(set_image_limit);
    com.bruxton.sidx.SIDXAcquire acquire = camera.AcquireOpen();

    try {
        acquireImageArchive(acquire);
    }
    catch (java.io.IOException exception) {
        System.out.println(exception.getMessage());
    }
    acquire.Close();
}

final static String archive_name = "ImageArchive1.tif";

private static void acquireImageArchive(com.bruxton.sidx.SIDXAcquire acquire) throws java.io.IOException {
    System.out.println("create file (ArchiveOpenNew) " + archive_name);
    acquire.ArchiveOpenNew(archive_ame, "TIFF", true);

    System.out.println("start acquisition (Start)");
    acquire.Start();

    long image_count = 0;
    long image_index = 0;

    // acquire.GetStatus()

    while (image_count < set_image_limit) {
     acquire.GetStatus();
        long current_image = acquire.ImageGetCount();
        long new_image_count = current_image - image_count;
        if (new_image_count > image_count) {
            acquire.ArchiveWrite(image_index, new_image_count);
            image_count = current_image;
            image_index = current_image;
        }
    }
```

```
        acquire.Stop();
        System.out.println("stop acquisition (Stop), image acquired " + image_count);
    }
}
```

SetParametersAndDisplay

To check if a specific kind of parameter can be set for the camera, for example intensifier gain, first check the setting type, if it returns NONE the parameter is not supported by the camera. The acquired images are read into an application.

```
public class SetParametersAndDisplay {
    public static void main (String[] args) {
        try {
            System.out.println("start");
            test();
            System.out.println("done");
        }
        catch (java.lang.Exception exception) {
            System.out.println(exception.getMessage());
        }
    }

    private static void test() throws java.io.IOException {
     com.bruxton.sidx.SIDXRootFactory root_create = new com.bruxton.sidx.SIDXRootFactory();
        String license = "";
     com.bruxton.sidx.SIDXRoot root = root_create.Open(license);

        root.CameraScan();
         // The first connected camera has the index '0'.
        String camera_name = root.CameraScanGetName(0);
        System.out.println("open camera (CameraOpenName) " + camera_name);
        com.bruxton.sidx.SIDXCamera camera = root.CameraOpenName(camera_name);

        try {
            cameraSetParameters(camera);
        }
        catch (java.lang.Exception exception) {
            System.out.println(exception.getMessage());
        }
        camera.Close();
        root.Close();
    }

    final static int set_binning_x = 1;
    final static int set_binning_y = 1;
    final static double set_exposure_duration = 0.01;
    final static double set_analog_gain = 1.0;
    final static int set_em_gain = 1;
    final static double set_intensifier_gain = 1.0;
    final static long set_image_limit = 100;

    final static com.bruxton.sidx.SIDXSettingType setting_type_none = com.bruxton.sidx.SIDXSettingType.NONE;
    final static com.bruxton.sidx.SIDXTriggerInMode trigger_mode = com.bruxton.sidx.SIDXTriggerInMode.ALWAYS;

    private static void cameraSetParameters(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
        com.bruxton.sidx.SIDXBinning set_binning = new com.bruxton.sidx.SIDXBinning(set_binning_x, set_binning_y);
        System.out.println("set binning (BinningSet) x " + set_binning_x + ", y " + set_binning_y);
```

```
        camera.BinningSet(set_binning);

        System.out.println("set exposure (ExposeSet) " + set_exposure_duration + " seconds");
        camera.ExposeSet(set_exposure_duration);

        if (camera.GainGetType() != setting_type_none) {
            System.out.println("set gain (GainSet) " + set_analog_gain);
            camera.GainSet(set_analog_gain);
        }

        if (camera.EMGainGetType() != setting_type_none) {
            System.out.println("set em gain (EMGainSet) " + set_em_gain);
            camera.EMGainSet(set_em_gain);
        }

        if (camera.IntensifierGetType() != setting_type_none) {
            System.out.println("set intensifier gain (IntensifierSet) " + set_intensifier_gain);
            camera.IntensifierSet(set_intensifier_gain);
        }

        if (camera.TriggerModeExists(trigger_mode)) {
            System.out.println("set trigger input (TriggerModeSet) " + trigger_mode);
            camera.TriggerModeSet(trigger_mode);
        }

        System.out.println("set image limit (AcquireImageSetLimit) " + set_image_limit);
        camera.AcquireImageSetLimit(set_image_limit);
        com.bruxton.sidx.SIDXAcquire acquire = camera.AcquireOpen();

        try {
            acquireImageDisplay(acquire);
        }
        catch (java.lang.Exception exception) {
            System.out.println(exception.getMessage());
        }
        acquire.Close();
    }

    private static void acquireImageDisplay(com.bruxton.sidx.SIDXAcquire acquire) throws java.io.IOException {
        System.out.println("start acquisition (Start)");
        acquire.Start();

        long image_count = acquire.ImageGetCount();
        int image_pixels = acquire.PixelGetCount().GetCountX() *
      acquire.PixelGetCount().GetCountY();
        boolean acquiring = true;
        while (acquiring) {
         acquiring = acquire.GetStatus();
            long current_image = acquire.ImageGetCount();
           long new_image_count = current_image - image_count;
            if (new_image_count > 0) {
                acquire.ReadSetPosition(current_image - 1);
                short[] image_buffer = new short[image_pixels];
                acquire.Read(1, image_buffer);
                image_count = current_image;
            }
        }

        acquire.Stop();
        System.out.println("stop acquisition (Stop), image acquired " + image_count);
    }
}
```

1.2 Enumeration

Obtain parameter information based on a count.

CameraScan

Find camera drivers connected to your system.

```
public class CameraScan {
   public static void main (String[] args) {
      try {
         System.out.println("start");
         test();
         System.out.println("done");
      }
      catch (java.io.IOException exception) {
         System.out.println(exception.getMessage());
      }
   }

   private static void test() throws java.io.IOException {
      com.bruxton.sidx.SIDXRootFactory root_create = new com.bruxton.sidx.SIDXRootFactory();
      String license = "";
      com.bruxton.sidx.SIDXRoot root = root_create.Open(license);

      root.CameraScan();

      // Obtain a report about the status of the camera drivers SIDX supports.
      String scan_report = root.CameraScanGetReport();
      System.out.println("camera scan report (CameraScanGetReport) " + scan_report);

      int camera_count = root.CameraScanGetCount();
      System.out.println("camera count (CameraScanGetCount) " + camera_count);
      if (camera_count > 0) {
         for (int index = 0; index < camera_count; ++index) {
            String camera_name = root.CameraScanGetName(index);
            String camera_label = root.CameraScanGetLabel(index);
            System.out.println("camera index " + index + " name " + camera_name + " label " + camera_label);
         }

         int camera_open_index = 0;
         String camera_name = root.CameraScanGetName(camera_open_index);
         com.bruxton.sidx.SIDXCamera camera = root.CameraOpenName(camera_name);

         // Conduct camera related operations.

         camera.Close();
      }
      root.Close();
   }
}
```

Operate

Operate modes determine the parameter values available.

```
public class Operate {
   public static void main (String[] args) {
      try {
         System.out.println("start");
         test();
         System.out.println("done");
      }
      catch (java.io.IOException exception) {
         System.out.println(exception.getMessage());
      }
   }

   private static void test() throws java.io.IOException {
      com.bruxton.sidx.SIDXRootFactory root_create = new com.bruxton.sidx.SIDXRootFactory();
      String license = "";
      com.bruxton.sidx.SIDXRoot root = root_create.Open(license);

      root.CameraScan();
      // The first connected camera has the index '0'.
      String camera_name = root.CameraScanGetName(0);
      System.out.println("open camera (CameraOpenName) " + camera_name);
      com.bruxton.sidx.SIDXCamera camera = root.CameraOpenName(camera_name);

      try {
         operateModes(camera);
      }
      catch (java.io.IOException exception) {
         System.out.println(exception.getMessage());
      }
      camera.Close();
      root.Close();
   }

   private static void operateModes(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
      int item_count = camera.OperateItemGetCount();
      System.out.println("operation mode item count (OperateItemGetCount) " + item_count);
      for (int item = 0; item < item_count; ++item) {
         String description = camera.OperateItemGetLocal(item);
         System.out.println(item + " description (OperateItemGetLocal) " + description);

         camera.OperateItemSet(item);
         String operation_name = camera.OperateGet();
         System.out.println("operation name (OperateGet) " + operation_name);
         // Camera parameters are dependent on operation mode.
      }
   }
}
```

Binning

Depending on the camera, binning of the x and y axis can be controlled separately or together.

```
public class Binning {
   public static void main (String[] args) {
      try {
         System.out.println("start");
         test();
         System.out.println("done");
      }
      catch (java.io.IOException exception) {
         System.out.println(exception.getMessage());
      }
   }

   private static void test() throws java.io.IOException {
      com.bruxton.sidx.SIDXRootFactory root_create = new com.bruxton.sidx.SIDXRootFactory();
      String license = "";
      com.bruxton.sidx.SIDXRoot root = root_create.Open(license);

      root.CameraScan();
      // The first connected camera has the index '0'
      String camera_name = root.CameraScanGetName(0);
      System.out.println("open camera (CameraOpenName) " + camera_name);
      com.bruxton.sidx.SIDXCamera camera = root.CameraOpenName(camera_name);

      try {
         binningType(camera);
      }
      catch (java.io.IOException exception) {
         System.out.println(exception.getMessage());
      }
      camera.Close();
      root.Close();
   }

   private static void binningType(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
      // Find out the binning readout behavior.
      com.bruxton.sidx.SIDXSettingType setting_type = camera.BinningGetType();
      switch (setting_type) {
      case LIST:
         binningList(camera);
         break;
      case NONE:
         binningNone(camera);
         break;
      default:
         break;
      }
   }

   private static void binningList(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
      int item_count = camera.BinningItemGetCount();
      System.out.println("binning type (BinningGetType) LIST has item count (BinningItemGetCount) " + item_count);
      for (int item = 0; item < item_count; ++item) {
         com.bruxton.sidx.SIDXBinning values = camera.BinningItemGetEntry(item);
         int x = values.GetX();
         int y = values.GetY();
         String description = camera.BinningItemGetLocal(item);
         System.out.println(item + " (BinningItemGetEntry) x " + x + ", y " + y + " (BinningItemGetLocal) " + description);
      }
   }

   private static void binningNone(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
      System.out.println("binning type (BinningGetType) NONE, x axis values are separate from y axis values");
```

```
        binningXType(camera);
        binningY(camera);
    }

    private static void binningXType(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
        com.bruxton.sidx.SIDXSettingType x_setting_type = camera.BinningXGetType();
        switch (x_setting_type) {
        case LIST:
            binningXList(camera);
            break;
        case INTEGER:
            binningXInteger(camera);
            break;
        case NONE:
            System.out.println("x axis binning type (BinningXGetType) NONE, axis values are co-dependent");
            binningList(camera);
            break;
        default:
            break;
        }
    }

    private static void binningXList(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
        int x_item_count = camera.BinningXItemGetCount();
        System.out.println("x axis binning type (BinningXGetType) LIST has item count (BinningXItemGetCount) " + x_item_count);
        for (int x_item = 0; x_item < x_item_count; ++x_item) {
            int x = camera.BinningXItemGetEntry(x_item);
            String x_description = camera.BinningXItemGetLocal(x_item);
            System.out.println(x_item + " (BinningXItemGetEntry) " + x + " (BinningXItemGetLocal) " + x_description);
        }
    }

    private static void binningXInteger(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
        int x_maximum = camera.BinningXGetLimit();
        System.out.println("x axis binning type (BinningXGetTyp) INTEGER range (BinningXGetLimit) 1 - " + x_maximum);
    }

    private static void binningY(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
        int y_maximum = camera.BinningYGetLimit();
        System.out.println("y axis binning range (BinningYGetLimit) 1 - " + y_maximum);
    }
}
```

Gain

Determine if analog gain is supported by your camera.

```
public class Gain {
    public static void main (String[] args) {
        try {
            System.out.println("start");
            test();
            System.out.println("done");
        }
        catch (java.io.IOException exception) {
            System.out.println(exception.getMessage());
        }
    }
```

```
public static void test() throws java.io.IOException {
    com.bruxton.sidx.SIDXRootFactory root_create = new com.bruxton.sidx.SIDXRootFactory();
    String license = "";
    com.bruxton.sidx.SIDXRoot root = root_create.Open(license);

    root.CameraScan();
    // The first connected camera has the index '0'.
    String camera_name = root.CameraScanGetName(0);
    System.out.println("open camera (CameraOpenName) " + camera_name);
    com.bruxton.sidx.SIDXCamera camera = root.CameraOpenName(camera_name);

    try {
        gainType(camera);
    }
    catch (java.io.IOException exception) {
        System.out.println(exception.getMessage());
    }
    camera.Close();
    root.Close();
}

private static void gainType(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
    com.bruxton.sidx.SIDXSettingType setting_type = camera.GainGetType();
    switch (setting_type) {
    case LIST:
        gainList(camera);
        break;
    case REAL:
        gainReal(camera);
        break;
    case NONE:
        System.out.println("gain type (GainGetType) " + setting_type + ", no analog gain for the camera");
        break;
    default:
        System.out.println("unknown gain type (GainGetType) " + setting_type);
        break;
    }
}

public static void gainList(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
    int item_count = camera.GainItemGetCount();
    System.out.println("gain type (GainGetType) LIST has item count (GainItemGetCount) " + item_count);
    for ( int item = 0; item < item_count; ++item) {
        double gain = camera.GainItemGetEntry(item);
        String description = camera.GainItemGetLocal(item);
        System.out.println(item + " (GainItemGetEntry) " + gain + " (GainItemGetLocal) " + description);
    }
    gainCommon(camera);
}

public static void gainReal(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
    com.bruxton.sidx.SIDXRangeReal gain_range = camera.GainGetRange();
    double minimum = gain_range.GetMinimum();
    double maximum = gain_range.GetMaximum();
    System.out.println("gain type (GainGetType) REAL, range (GainGetRange) " + minimum + " - " + maximum);
    gainCommon(camera);
}

public static void gainCommon(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
    String label = camera.GainGetLabel();
    String unit = camera.GainGetUnit();
    double driver_value = camera.GainGetValue();
    System.out.println("gain label (GainGetLabel) " + label + ", value (GainGetValue) " + driver_value + ", unit (GainGetUnit) " + unit);
```

```
    }

}
```

Shutter

Find the available shutter modes.

```
public class Shutter {
    public static void main(String[] args) {
        try {
            System.out.println("start");
            test();
            System.out.println("done");
        }
        catch (java.io.IOException exception) {
            System.out.println(exception.getMessage());
        }
    }

    private static void test() throws java.io.IOException {
        com.bruxton.sidx.SIDXRootFactory root_create = new com.bruxton.sidx.SIDXRootFactory();
        String license = "";
        com.bruxton.sidx.SIDXRoot root = root_create.Open(license);

        root.CameraScan();
        // The first connected camera has the index '0'
        String camera_name = root.CameraScanGetName(0);
        System.out.println("open camera (CameraOpenName) " + camera_name);
        com.bruxton.sidx.SIDXCamera camera = root.CameraOpenName(camera_name);

        com.bruxton.sidx.SIDXShutterMode mode;
        com.bruxton.sidx.SIDXShutterMode[] count = com.bruxton.sidx.SIDXShutterMode.values();
        int index;
        for (index = 0; index < count.length; ++index) {
            mode = count[index];
            boolean exists = camera.ShutterExists(mode);
            if (exists)
                System.out.println("available shutter mode (ShutterExists) " + mode);
        }

        camera.Close();
        root.Close();
    }
}
```

TriggerMode

Find the available trigger input modes.

```
public class TriggerMode {
    public static void main(String[] args) {
        try {
```

```
            System.out.println("start");
            test();
            System.out.println("done");
        }
        catch (java.io.IOException exception) {
            System.out.println(exception.getMessage());
        }
    }

    private static void test() throws java.io.IOException {
        com.bruxton.sidx.SIDXRootFactory root_create = new com.bruxton.sidx.SIDXRootFactory();
        String license = "";
        com.bruxton.sidx.SIDXRoot root = root_create.Open(license);

        root.CameraScan();
        // The first connected camera has the index '0'
        String camera_name = root.CameraScanGetName(0);
        System.out.println("open camera (CameraOpenName) " + camera_name);
        com.bruxton.sidx.SIDXCamera camera = root.CameraOpenName(camera_name);

        com.bruxton.sidx.SIDXTriggerInMode mode;
        com.bruxton.sidx.SIDXTriggerInMode[] count = com.bruxton.sidx.SIDXTriggerInMode.values();
        int index;
        for (index = 0; index < count.length; ++index) {
            mode = count[index];
            boolean exists = camera.TriggerModeExists(mode);
            if (exists)
                System.out.println("available trigger mode (TriggerModeExists) " + mode);
        }

        camera.Close();
        root.Close();
    }
}
```

TriggerSignal

Find the avilable trigger signal modes.

```
public class TriggerSignal {
    public static void main(String[] args) {
        try {
            System.out.println("start");
            test();
            System.out.println("done");
        }
        catch (java.io.IOException exception) {
            System.out.println(exception.getMessage());
        }
    }

    private static void test() throws java.io.IOException {
        com.bruxton.sidx.SIDXRootFactory root_create = new com.bruxton.sidx.SIDXRootFactory();
        String license = "";
        com.bruxton.sidx.SIDXRoot root = root_create.Open(license);

        root.CameraScan();
        // The first connected camera has the index '0'
```

```
        String camera_name = root.CameraScanGetName(0);
        System.out.println("open camera (CameraOpenName) " + camera_name);
        com.bruxton.sidx.SIDXCamera camera = root.CameraOpenName(camera_name);

        triggerSignal(camera);

        camera.Close();
        root.Close();
    }

    private static void triggerSignal(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
        com.bruxton.sidx.SIDXSignalActiveMode mode;
        com.bruxton.sidx.SIDXSignalActiveMode[] count = com.bruxton.sidx.SIDXSignalActiveMode.values();
        int index;
        for (index = 0; index < count.length; ++index) {
            mode = count[index];
            boolean exists = camera.TriggerSignalExists(mode);
            if (exists)
                System.out.println("available trigger signal (TriggerSignalExists) " + mode);
        }
    }
}
```

1.3 Stage Control

Control a motorized stage.

StageGetLimits

Move the stage to the positive and negative X axis and Y axis limits.

```
public class StageGetLimits {
   public static void main(String[] args) {
      try{
         System.out.println("start");
         test();
         System.out.println("done");
      }
      catch (java.lang.Exception exception) {
         System.out.println("Exception (Exception) " + exception.getMessage());
      }
   }

   final static double ABSOLUTE_X_POSITION = 0.00000;
   final static double ABSOLUTE_Y_POSITION = 0.0000;
   final static double ABSOLUTE_Z_POSITION = 0.0000;
   final static double CONSTANT_MOVE_SPEED_POSITIVE = 0.0005;
   final static double CONSTANT_MOVE_SPEED_NEGATIVE = -0.0005;

   final static double CONSTANT_SET_ACCELERATION = 0.75; // fraction
   final static double CONSTANT_SET_SPEED = 0.001;
   final static double CONSTANT_SET_BACKLASH = 0.001;

   final static int CONSTANT_SLEEP_DURATION = 1; // millisecond

   private static void test() throws Exception {
      com.bruxton.sidx.SIDXRootFactory root_create = new com.bruxton.sidx.SIDXRootFactory();
      String license = "";
      com.bruxton.sidx.SIDXRoot root = root_create.Open(license);

      // The port number varies depending on where the stage is connected.
      String name = "Applied Scientific Instrumentation";
      String port = "COM8";
      com.bruxton.sidx.SIDXStage stage = root.StageOpen(name, port);
      System.out.println("open stage from port (StageOpen) " + port);

      String stage_name = stage.GetName();
      System.out.println("open stage (GetName) " + stage_name);

      stageSetUp(stage);

      resetAbsolutePosition(stage);
      stageLimits(stage);

      stage.Close();
      root.Close();
   }

   private static void stageSetUp(com.bruxton.sidx.SIDXStage stage) throws java.io.IOException {
      double xy_acceleration_limit = stage.AccelerateGetLimitXY();
```

```
        System.out.println("get acceleration limit (AccelerateGetLimitXY) " + String.valueOf(xy_acceleration_limit));

        stage.AccelerateSetXY(CONSTANT_SET_ACCELERATION);

        double xy_accelerate_get = stage.AccelerateGetXY();
        System.out.println("set acceleration fraction (AccelerateGetXY) " + String.valueOf(xy_accelerate_get));

        stage.SpeedSetXY(CONSTANT_SET_SPEED);

        double speed = stage.SpeedGetXY();
        System.out.println("set speed (SpeedGetXY) " + speed);

        if (stage.BacklashExists()) {
            stage.BacklashSetXY(CONSTANT_SET_BACKLASH);

            double backlash = stage.BacklashGetXY();
            System.out.println("set backlash (BacklashGetXY) " + backlash);
        }
    }

    private static void resetAbsolutePosition(com.bruxton.sidx.SIDXStage stage) throws java.io.IOException, InterruptedException {
//        stage.MovePositionXYZ(ABSOLUTE_X_POSITION, ABSOLUTE_Y_POSITION, ABSOLUTE_Z_POSITION);
//        boolean moving = true;
//        while (moving) {
//            stage.MovingQuery();
//            moving = stage.MovingIsXYZ();
//            Thread.sleep(CONSTANT_SLEEP_DURATION);
//        }
//
//        System.out.println("reset to absolute position");
//        getCurrentPosition(stage);

     stage.MovePositionXY(ABSOLUTE_X_POSITION, ABSOLUTE_Y_POSITION);
     boolean moving = true;
     while (moving) {
         stage.MovingQuery();
         moving = stage.MovingIsXY();
         Thread.sleep(CONSTANT_SLEEP_DURATION);
     }

     System.out.println("reset to absolute position");
     getCurrentPosition(stage);
 }

    private static void getCurrentPosition(com.bruxton.sidx.SIDXStage stage) throws java.io.IOException {
        stage.PositionQuery();
        com.bruxton.sidx.SIDXStageCoordinates position = stage.PositionGet();
        double position_x = position.GetPositionX();
        double position_y = position.GetPositionY();
        double position_z = position.GetPositionZ();

        System.ut.println("get position (PositionGet) x " + position_x + " y " + position_y + " z " + position_z);
    }

    private static void stageLimits(com.bruxton.sidx.SIDXStage stage) throws java.io.IOException, InterruptedException {
        boolean moving;
        do {
            stage.MoveSpeedXY(CONSTANT_MOVE_SPEED_POSITIVE, CONSTANT_MOVE_SPEED_POSITIVE);

            stage.MovingQuery();
            moving = stage.MovingIsXY();
            if (moving)
                Thread.sleep(CONSTANT_SLEEP_DURATION);
```

```
    stage.LimitQuery();
    if (stage.LimitIsXPlus() && stage.LimitIsYPlus()) {
        stage.MoveStopXYZ();
        break;
    }
} while(moving);

System.out.println("stage limits x0, y0");
getCurrentPosition(stage);

do {
    stage.MoveSpeedXY(0, CONSTANT_MOVE_SPEED_NEGATIVE);

    stage.MovingQuery();
    moving = stage.MovingIsXY();
    if (moving)
        Thread.sleep(CONSTANT_SLEEP_DURATION);

    stage.LimitQuery();
    if (stage.LimitIsYMinus()) {
        stage.MoveStopXYZ();
        break;
    }
} while(moving);

System.out.println("stage limits x0, y1");
getCurrentPosition(stage);

do {
    stage.MoveSpeedXY(CONSTANT_MOVE_SPEED_NEGATIVE, 0);

    stage.MovingQuery();
    moving = stage.MovingIsXY();
    if (moving)
        Thread.sleep(CONSTANT_SLEEP_DURATION);

    stage.LimitQuery();
    if (stage.LimitIsXMinus()) {
        stage.MoveStopXYZ();
        break;
    }
} while(moving);

System.out.println("stage limits x1, y1");
getCurrentPosition(stage);

do {
    stage.MoveSpeedXY(0, CONSTANT_MOVE_SPEED_POSITIVE);

    stage.MovingQuery();
    moving = stage.MovingIsXY();
    if (moving)
        Thread.sleep(CONSTANT_SLEEP_DURATION);

    stage.LimitQuery();
    if (stage.LimitIsYPlus()) {
        stage.MoveStopXYZ();
        break;
    }
} while(moving);

System.out.println("stage limits x1, y0");
getCurrentPosition(stage);
```

```
        do {
            stage.MoveSpeedXY(CONSTANT_MOVE_SPEED_POSITIVE, 0);

            stage.MovingQuery();
            moving = stage.MovingIsXY();
            if (moving)
                Thread.sleep(CONSTANT_SLEEP_DURATION);

            stage.LimitQuery();
            if (stage.LimitIsXPlus()) {
                stage.MoveStopXYZ();
                break;
            }
        } while(moving);

        System.out.println("stage limits x0, y0");
        getCurrentPosition(stage);
    }
}
```

StageStepAndRepeat

Move the stage to absolute X axis and Y axis positions. Image every ***x*** distance.

```
public class StageStepAndRepeat {
    public static void main(String[] args) {
        try{
            System.out.println("start");
            test();
            System.out.println("done");
        }
        catch (java.lang.Exception exception) {
            System.out.println("Exception (Exception) " + exception.getMessage());
        }
    }

    final static double ABSOLUTE_X_POSITION = 0.0000; // absolute position in meters
    final static double ABSOLUTE_Y_POSITION = 0.0000; // absolute position in meters

    final static double TARGET_X_POSITION = 0.0050; // absolute position in meters
    final static double TARGET_Y_POSITION = 0.0050; // absolute position in meters

    final static double MOVE_DISTANCE_POSITIVE = 0.001; // relative position in meters
    final static double MOVE_DISTANCE_NEGATIVE = -0.001; // relative position in meters

    private static void test() throws Exception {
        com.bruxton.sidx.SIDXRootFactory root_create = new com.bruxton.sidx.SIDXRootFactory();
        String license = "";
        com.bruxton.sidx.SIDXRoot root = root_create.Open(license);

        root.CameraScan();
        // The first connected camera has the index '0'
        String camera_name = root.CameraScanGetName(0);
        System.out.println("open camera (root.CameraOpenName) " + camera_name);
        com.bruxton.sidx.SIDXCamera camera = root.CameraOpenName(camera_name);

        // The port number varies depending on where the stage is connected.
        String name = "Prior Scientific";
```

```
    String port = "COM9";
    com.bruxton.sidx.SIDXStage stage = root.StageOpen(name, port);
    System.out.println("open stage from port (root.StageOpen) " + port);

    stageSetUp(stage);
    cameraSetUp(camera);

    com.bruxton.sidx.SIDXAcquire acquire = camera.AcquireOpen();
    String archive_name = "ImageArchiveStepAndRepeat.tiff";
    acquire.ArchiveOpenNew(archive_name, "TIFF", true);

    double y_position;
    do {
       moveXAxis(acquire, stage, MOVE_DISTANCE_POSITIVE, TARGET_X_POSITION);

       stage.PositionQuery();
       y_position = stage.PositionGetY();
       if (y_position >= TARGET_Y_POSITION)
        break;

       double move_y_position = y_position + MOVE_DISTANCE_POSITIVE;
       moveYAxis(acquire, stage, MOVE_DISTANCE_POSITIVE, move_y_position);

       moveXAxis(acquire, stage, MOVE_DISTANCE_NEGATIVE, ABSOLUTE_X_POSITION);

       stage.PositionQuery();
       y_position = stage.PositionGetY();
       if (y_position >= TARGET_Y_POSITION)
        break;

       move_y_position = y_position + MOVE_DISTANCE_POSITIVE;
       moveYAxis(acquire, stage, MOVE_DISTANCE_POSITIVE, move_y_position);

    } while(y_position <= TARGET_Y_POSITION);

    acquire.Close();
    stage.Close();
    camera.Close();
    root.Close();
}

final static double CONSTANT_SET_ACCELERATION = 0.1; // fraction
final static int CONSTANT_SLEEP_DURATION = 1; // milliseconds

private static void stageSetUp(com.bruxton.sidx.SIDXStage stage) throws java.io.IOException, InterruptedException {
    String stage_name = stage.GetName();
    System.out.println("open stage (stage.GetName) " + stage_name);

    stage.AccelerateSetXY(CONSTANT_SET_ACCELERATION);

    System.out.println("move to absolute position (stage.MovePositionXY)");
    stage.MovePositionXY(ABSOLUTE_X_POSITION, ABSOLUTE_Y_POSITION);
    stage.MovingQuery();
    while (stage.MovingIsXY()) {
       stage.MovingQuery();
       Thread.sleep(CONSTANT_SLEEP_DURATION);
    }

    stage.PositionQuery();
    System.out.println("get position x (stage.PositionGetX) " + String.valueOf(stage.PositionGetX()));
    System.out.println("get positiony (stage.PositionGetY) " + String.valueOf(stage.PositionGetY()));
}

final static double COSTANT_SET_EXPOSURE = 0.1;
```

```
final static long CONSTANT_SET_IMAGE_LIMIT = 1;

private static void cameraSetUp(com.bruxton.sidx.SIDXCamera camera) throws java.io.IOException {
    camera.ExposeSet(COSTANT_SET_EXPOSURE);

    camera.AcquireImageSetLimit(CONSTANT_SET_IMAGE_LIMIT);
    System.out.println("set image limit (camera.AcquireImageSetLimit) " + String.valueOf(CONSTANT_SET_IMAGE_LIMIT));
}

private static void acquireSingleShot(com.bruxton.sidx.SIDXAcquire acquire) throws java.io.IOException {
    acquire.Start();

    long image_count = acquire.ImageGetCount();
    long image_index = image_count;

    while (image_count < CONSTANT_SET_IMAGE_LIMIT) {
        acquire.GetStatus();
        long current_image_count = acquire.ImageGetCount();
        long new_image_count = current_image_count - image_count;
        if (new_image_count > image_count) {

            acquire.ArchiveWrite(image_index, new_image_count);

            image_count = current_image_count;
            image_index = current_image_count;
        }
    }
    acquire.Stop();
}

private static void moveXAxis(
    com.bruxton.sidx.SIDXAcquire acquire,
    com.bruxton.sidx.SIDXStage stage,
    double move_length,
    double target_position)
throws java.io.IOException, InterruptedException {
    double current_x_position;
    boolean direction_plus = true;
    do {
        acquireSingleShot(acquire);

        stage.PositionQuery();
        double stagnant_y_position = stage.PositionGetY();
        double previous_x_position = stage.PositionGetX();
        current_x_position = previous_x_position + move_length;

        stage.MovePositionXY(current_x_position, stagnant_y_position);
        stage.MovingQuery();
        while (stage.MovingIsXY()) {
            stage.MovingQuery();
            Thread.sleep(CONSTANT_SLEEP_DURATION);
        }

        stage.LimitQuery();
        if (stage.LimitIsXY())
            break;

        if (current_x_position < previous_x_position)
            direction_plus = false;
    } while(approachingDestination(direction_plus, current_x_position, target_position));

    stage.PositionQuery();
    System.out.println("current stage position x axis " + String.valueOf(stage.PositionGetX()));
}
```

```
    private static void moveYAxis(
        com.bruxton.sidx.SIDXAcquire acquire,
        com.bruxton.sidx.SIDXStage stage,
        double move_length,
        double target_position)
    throws java.io.IOException, InterruptedException {
     double current_y_position;
        boolean direction_plus = true;
        do {
            acquireSingleShot(acquire);

            stage.PositionQuery();
            double stagnant_x_position = stage.PositionGetX();
            double previous_y_position = stage.PositionGetY();
            current_y_position = previous_y_position + move_length;

            stage.MovePositionXY(stagnant_x_position, current_y_position);
            stage.MovingQuery();
            while (stage.MovingIsXY()) {
                stage.MovingQuery();
                Thread.sleep(CONSTANT_SLEEP_DURATION);
            }

            stage.LimitQuery();
            if (stage.LimitIsXY())
                break;

            if (current_y_position < previous_y_position)
                direction_plus = false;
        } while (approachingDestination(direction_plus, current_y_position, target_position));

        stage.PositionQuery();
        System.out.println("current stage position y axis " + String.valueOf(stage.PositionGetY()));
    }

    private static boolean approachingDestination(boolean movement, double current_position, double target_position)
    {
        if (movement) {
            if (current_position >= target_position)
                return false;
        }
        else {
            if (current_position <= target_position)
                return false;
        }

        return true;
    }
}
```

2. Camera

2.1 Andor Technology

SIDXDevice…

Specification

SIDXDevice…	Returns
…ActionGetCount	0
…DriverGetDescription	"Andor Technology *version*"
…DriverGetName	"Andor Technology"
…DriverGetType	SIDX_DRIVER_TYPE_ANDOR_TECHNOLOGY
…ExtraGetCount	12
…GetDescription	"*SDK version, device driver version, head model, camera serial number*"
…GetName	"Andor Technology"
…GetLabel	"Andor Technology *model*
…PortGetCount	0

Notes

1. The value returned by …ActionGetCount is always 0 (zero), because the Andor Technology camera has no device-specific actions.
2. The driver description text string returned by …DriverGetDescription contains the driver *version*, for example "Andor Technology 2.86.30000", where "2.86.30000" represents the *version*.
3. The description text string returned by …GetDescription contains manufacturer hardware information about the Andor Technology camera and the Andor Technology board.
4. The label text string returned by …GetLabel contains the camera *model*, for example "Andor Technology Luca", where "Luca" represents the *model*.
5. The value returned by …PortGetCount is always 0 (zero), because the Andor Technology camera has no ancillary analog or digital inputs or outputs.

SIDXDeviceExtra…

"BaselineClamp"

Baseline clamp is used to adjust for temperature fluctuations in the electronics across a kinetic series. Refer to the camera manual for more description of the function. True disables the baseline clamp, false enables the baseline clamp.

SIDXDeviceExtra…	**Returns**
…GetName	"BaselineClamp"
…GetType	SIDX_SETTING_TYPE_BOOLEAN
…IsSettable	true

"VerticalShiftSpeed"

Set the vertical pixel shift speeds. Please refer to Andor Technology documentation for further discussion.

SIDXDeviceExtra…	**Returns**
…GetName	"VerticalShiftSpeed"
…GetType	SIDX_SETTING_TYPE_LIST
…IsSettable	true
…ListGetCount	*driver dependent*

"FastKineticVerticalShiftSpeed"

The value is used for controlling the vertical pixel shift speed in the special readout mode to allow an extremely fast sequence of images to be captured. Refer to Andor Technology documentation for fast kinetics mode. The vertical shift speed is in microseconds per pixel shift.

SIDXDeviceExtra…	**Returns**
…GetName	"FastKineticVerticalShiftSpeed"
…GetType	SIDX_SETTING_TYPE_LIST
…GetUnit	microseconds per pixel shift
…IsSettable	true
…ListGetCount	*driver dependent*

"FrameTransfer"

If set to true, the camera acquires images using frame transfer mode. If false, full-frame mode.

SIDXDeviceExtra…	Returns
…GetName	"FrameTransfer"
…GetType	SIDX_SETTING_TYPE_BOOLEAN
…IsSettable	true

Note

- The exposure time cannot be less than the readout time (the shift of image data from the image area to storage area cannot happen before the readout is completed).
- In frame transfer mode, the parameters used for kinetic series mode (accumulation cycle time, kinetic cycle time, accumulation count and so on) are irrelevant and not used.
- The frame rate in frame transfer mode is determined by the exposure time and the time to shift the image under the storage area.

"PhotonCounting"

Photon counting is only availabe in iStar and iXon cameras (see Andor camera specification for more details). Photon counting counts the number of times the signal level for a particular pixel is between the minimum_threshold and maximum_threshold level. If there are 500 scans in the accumulation and a particular pixel is counted within the threshold 100 times, the photon counting value for the pixel will be 100. The string parameter enables or disables photon counting and sets the minimum_threshold and maximum_threshold level. For example, to enable photon counting set "true, 20, 200".

SIDXDeviceExtra…	Returns
…GetName	"PhotonCounting"
…GetType	SIDX_SETTING_TYPE_STRING
…IsSettable	true
…StringSet	"*counting, minimum_threshold, maximum_threshold*"

The available string value combinations are listed in the following table.

counting	minimum_threshold	maximum_threshold
"false"	*ignored*	*ignored*
"true"	*integer value*	*integer value*

"SkipCleaning"

True skips the cleaning cycle, false runs the cleaning cycle.

SIDXDeviceExtra…	Returns
…GetName	"SkipCleaning"
…GetType	SIDX_SETTING_TYPE_BOOLEAN
…IsSettable	true

"VerticalClockVoltage"

Set the vertical pixel shift speeds. Please refer to Andor Technology documentation for further discussion.

SIDXDeviceExtra…	Returns
…GetName	"VerticalClockVoltage"
…GetType	SIDX_SETTING_TYPE_LIST
…IsSettable	true
…ListGetCount	5
…ListGetLocal (0)	"Normal"
…ListGetLocal (1)	"+1"
…ListGetLocal (2)	"+2"
…ListGetLocal (3)	"+3"
…ListGetLocal (3)	"+4"

"AccumulationCycleTime"

The parameter should be set to zero (0) for fastest acquisition, limited only the the exposure time.

SIDXDeviceExtra…	Returns
…GetName	"AccumulationCycleTime"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	*available at runtime*

"KineticCycleTime"

The parameter should be set to zero (0) for fastest acquisition, limited only the the exposure time.

SIDXDeviceExtra…	Returns
…GetName	"KineticCycleTime"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	*available at runtime*

"AccumulationCount"

The parameter should be set to zero (0) for fastest acquisition, limited only the the exposure time.

SIDXDeviceExtra…	Returns
…GetName	"AccumulationCount"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…RealGetRange	*available at runtime*

"ShutterCloseTime"

The parameter sets the time the shutter takes to close in seconds.

SIDXDeviceExtra…	Returns
…GetName	"ShutterCloseTime"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	*available at runtime*

"ShutterOpenTime"

The parameter sets the time the shutter takes to open in seconds.

SIDXDeviceExtra…	Returns
…GetName	"ShutterOpenTime"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	*available at runtime*

"ShutterSignalHigh"

The parameter controls how the output TTL signal is when opening the shutter. If it is to true, the output TTL signal is high when open the shutter. Otherwise low when open the shutter.

SIDXDeviceExtra…	Returns
…GetName	"ShutterSignalHigh"
…GetType	SIDX_SETTING_TYPE_BOOLEAN
…IsSettable	true

"ComplextImage"

The parameter controls if the "Tracks" settings are used for imaging. If it is set to true, the settings in "Tracks" are used for image acquisition. Otherwise not.

SIDXDeviceExtra…	Returns
…GetName	"ComplexImage"
…GetType	SIDX_SETTING_TYPE_BOOLEAN
…IsSettable	true

"Tracks"

The parameter defines the geometry of track(s) for imaging. Define the top and bottom row number starting from zero for each track separated by comma.

SIDXDeviceExtra…	Returns
…GetName	"Tracks"
…GetType	SIDX_SETTING_TYPE_STRING
…IsSettable	true
…StringSet	*top_1,bottom_1,top_2,bottom_2,...*

"FanMode"

To set the fan mode for either full or low, the fan control must be enabled. If the fan control is disabled, the fan is turned off.

SIDXDeviceExtra…	Returns
…GetName	"FanMode"
…GetType	SIDX_SETTING_TYPE_LIST
…IsSettable	true
…ListGetCount	2
…ListGetLocal (0)	"On full"
…ListGetLocal (1)	"On low"

SIDXCamera…

Specification

SIDXCamera…	Returns
…BinningGetType	SIDX_SETTING_TYPE_NONE
…BinningXGetType	SIDX_SETTING_TYPE_LIST
…BinningYGetLimit	*height*
…CoolingControlGet	SIDX_COOLING_CONTROL_GET_ONLY or SIDX_COOLING_CONTROL_GET_SET
…EMGainGetType	SIDX_SETTING_TYPE_INTEGER
…EMGainGetRange	*minimum* to *maximum*
…ExposeArrayGetSize	0
…ExposeGetRange	0 to *maximum*
…FanControlExists	true
…GainGetType	SIDX_SETTING_TYPE_LIST
…IntensifierGetType	SIDX_SETTING_TYPE_REAL or SIDX_SETTING_TYPE_NONE
…OperateItemGetCount	2
…OperateItemGetLocal (0)	"Default"
…OperateItemGetLocal (1)	"FastKinetic"
…ReadoutItemGetCount	1
…TemperatureExists	true

Notes

1. The y axis binning limit is the *height* of the camera sensor. The actual value is camera dependent.
2. The type returned by …CoolingControlGet is camera dependent. All cameras provide sensor temperature cooling information, however some also allow set control.
3. The *minimum* and *maximum* EMGain values are camera dependent.
4. The *maximum* exposure value is camera dependent.
5. The type returned by …IntensifierGetType is camera dependent. If the type returned is SIDX_SETTING_TYPE_REAL, a camera has intensifier gain and the range is between 0 to 255.

SIDXCameraShutterMode…

Use …ShutterModeExists to obtain the available shutter modes for a specific Andor Technology camera.

SIDXShutterMode	Available
SIDX_SHUTTER_MODE_OPEN_DURING_EXPOSURE	always
SIDX_SHUTTER_MODE_OPEN	always
SIDX_SHUTTER_MODE_CLOSE	always

SIDXCameraTriggerMode…

Use …TriggerModeExists to obtain the available trigger input modes for a specific Andor Technology camera.

SIDXTriggerInMode	Available
SIDX_TRIGGER_IN_MODE_ALWAYS	always
SIDX_TRIGGER_IN_MODE_EXPOSURE_START	always
SIDX_TRIGGER_IN_MODE_EXPOSURE_DURATION	camera dependent
SIDX_TRIGGER_IN_MODE_SEQUENCE_START	always

Setting

AcquisitionMode

- *type* sequence
- *range* model
- *default* image

If you are looking for getting the fastest possible frame rate from your Andor camera, you may want to use the "fast kinetic acquisition" mode. In this mode, the camera takes a certain number of images, and stores them on the camera rather than in the computer. After taking all the images, the camera reads the images out into the computer. As the result, it eliminates the readout time spent between each exposure in order to speed up the frame rate.

SIDX runs in this mode automatically when the settings meet the requirements. You will have to make sure that all images will fit into the CCD. For example, 64 discrete images can be stored using a CCD-chip with a height of 512 pixels and a sub area 8 rows high. Please notice that there are sometimes certain areas on the CCD-chip cannot be used for storing. Please check with the camera specification or with the camera vendor for details.

EMGainMode

- *type* list
- *range* driver
- *default* driver highest available
- *data* DAC 0 255, DAC 0 4095, Linear, Real EM gain

EM gain describes the gain amplification of an electron mutiplier. For a camera that supports EM gain, the user may choose one of the several EM gain modes to use. The EM gain mode defines the data range of the DAC settings that controls the EM gain. For example, in the default EM gain mode, the EM gain is controlled by DAC settings in the range 0 - 255. The EM gain is always set under a given EM gain mode. In some Andor cameras the settable EM gain range may vary with temperature; the application should call to find out the available EM gain range when the temperature setpoint is changed.

ReadoutMode

- *type* sequence
- *range* driver
- *default* image

"readout" is used to describe how the camera builds up the pixel data of an image. A "track" is a banded region on the CCD. It has a certain height (less than or equal to the CCD height) but always has the full CCD width. In the multi-track mode, the user may specify one or multiple tracks. Each track results in a net single charge per column. There are several ways to divid the CCD into bands: a) full CCD is one band, b) a single band located anywhere on the CCD, c) multiple bands evenly spaced, and d) multiple bands located anywhere on the CCD but without overlapping. Any Andor camera may have one or more of the capabilities. In order to run the camera in multi-track readout mode, the readout mode of the camera should be set to multi-track readout mode.

2.2 AVT

SIDXDevice…

Specification

SIDXDevice…	Returns
…ActionGetCount	1
…DriverGetDescription	"Allied Vision Technologies Universal API *version*"
…DriverGetName	"Allied Vision Technologies Universal API"
…DriverGetType	SIDX_DRIVER_TYPE_AVT
…ExtraGetCount	9
…GetDescription	"*serial number, micro controller version, FPGA version, order number*"
…GetName	"AVT *identification*"
…GetLabel	"AVT *model*"
…PortGetCount	0

Notes

1. The driver description text string returned by …DriverGetDescription contains the AVT driver ***version*** for example, "Allied Vision Technologies Universal API 1.2.3.4.", where "1.2.3.4." represents the ***version***.
2. The description text string returned by …GetDescription contains manufacturer hardware information about the AVT camera.
3. The name text string returned by …GetName contains the AVT ***identification***, where the ***identification*** is a decimal digit in the range greater than or equal to 0, for example "AVT 0".
4. The description text string returned by …GetLabel contains the AVT ***model***, for example "AVT Stringray" where "Stingray" represents the model.
5. The value returned by …PortGetCount is always 0 (zero), because AVT cameras have no ancillary analog or digital inputs or outputs.

SIDXDeviceAction…

"SoftReset"

FPGA reboot and bus reset.

SIDXDeviceExtra…

"AutoExposure"

The setting controls the target grey level. The string parameter contains values that determine the control of the setting. The exposure_value must be within the exposure_minimum and exposure_maximum range. For example, if you set "10, 0, 100", the exposure_value is 10 which must be between 0 and 100.

SIDXDeviceExtra…	**Returns**
…GetName	"AutoExposure"
…GetType	SIDX_SETTING_TYPE_STRING
…IsSettable	true
…StringSet	*exposure_value, exposure_minimum, exposure_maximum*

The available string value combinations are listed in the following table.

automatic_adjust	**exposure_value**	**exposure_minimum**	**exposure_maximum**
"true"	*ignored*	*ignored*	*ignored*
"false"	*current integer value*	*available at runtime*	*available at runtime*

"Brightness"

The setting controls the image brightness (black level).

SIDXDeviceExtra…	**Returns**
…GetName	"Brightness"
…GetType	SIDX_SETTING_TYPE_STRING
…IsSettable	true
…StringSet	*automatic_adjust, exposure_value, exposure_minimum, exposure_maximum*

The available string value combinations are listed in the following table.

automatic_adjust	**exposure_value**	**exposure_minimum**	**exposure_maximum**
"true"	*ignored*	*ignored*	*ignored*
"false"	*current integer value*	*available at runtime*	*available at runtime*

"CameraAcceptDelay"

Set the timeout length in seconds when wait for response.

SIDXDeviceExtra…	**Returns**
…GetName	"CameraAcceptDelay"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	*available at runtime*

"CycleTime"

Actual bus time in seconds. Currently not supported by SIDX.

SIDXDeviceExtra…	Returns
…GetName	"CycleTime"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	false
…RealGetRange	*available at runtime*

"DelayedIntegrationEnable"

Delay the leading edge of the integration enable output (Pin IntEna) in seconds, and the signal never goes down

SIDXDeviceExtra…	Returns
…GetName	"DelayedIntegrationEnable"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	*available at runtime*

"SerialIOConfiguration"

Configure the serial communication. As an example, to set baud_rate to 38400, parity to odd, stop_bit to 1, and IO_mode to receive and transmit, the string appears as "38400, Odd, 1, ReceiveTransmit".

SIDXDeviceExtra…	Returns
…GetName	"SerialIOConfiguration"
…GetType	SIDX_SETTING_TYPE_STRING
…IsSettable	true
…StringSet	*baud_rate, parity, stop_bit, IO_mode*

The available string value combinations are listed in the following table.

baud_rate	parity	stop_bit	IO_mode
"300"	"None"	"1"	"Disabled"
"600"	"Odd"	"1.5"	"Receive"
"1200"	"Even"	"2"	"Transmit"
"2400"			"ReceiveTransmit"
"4800"			
"9600"			
"19200"			
"38400"			
"57600"			
"115200"			
"230400"			

"TriggerDelay"

Set the delay of the trigger signal in seconds.

SIDXDeviceExtra…	Returns
…GetName	"TriggerDelay"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	*available at runtime*

SIDXCamera…

Specification

SIDXCamera…	Returns
…BinningGetType	SIDX_SETTING_TYPE_LIST
…CoolingControlGet	SIDX_COOLING_CONTROL_NONE
…EMGainGetType	SIDX_SETTING_TYPE_NONE
…ExposeArrayGetSize	0
…ExposeGetRange	0 to *maximum*
…FanControlExists	false
…GainGetType	SIDX_SETTING_TYPE_REAL or SIDX_SETTING_TYPE_NONE
…GainGetRange	*minimum* to *maximum*
…IntensifierGetType	SIDX_SETTING_TYPE_NONE
…OperationItemGetCount	*1*
…OperationItemGetLocal	*width_pixel* x *height_pixel*
…ReadoutItemGetCount	*camera dependent*
…TemperatureExists	false

Notes

1. The *maximum* exposure value returned by …ExposeGetRange is camera dependent.
2. If analog gain control is available for the given camera, the value returned by …GainGetType is SIDX_SETTING_TYPE_REAL. If analog gain control is not available, the value returned is SIDX_SETTING_TYPE_NONE.
3. The *minimum* and *maximum* gain values returned by …GainGetRange are camera dependent.
4. The text string returned by …OperateItemGetLocal contains the image *width_pixel* by *height_pixel* values that correspond to the operation mode. For example, index 0 for …OperateItemGetLocal could return "100x100".

2.3 Jenoptik

SIDXDevice…

Specification

SIDXDevice…	Returns
…ActionGetCount	0
…DriverGetDescription	"Jenoptik MexCam *version*"
…DriverGetName	"Jenoptik MexCam"
…DriverGetType	SIDX_DRIVER_TYPE_JENOPTIK
…ExtraGetCount	0 or greater
…GetDescription	*serial number*
…GetName	"Jenoptik *number*"
…GetLabel	"Jenoptik *model*"
…PortGetCount	0

Notes

1. The value returned by …ActionGetCount depends on the Jenopik camera. Currently there are no action controls available, however in the future action controls may be added to SIDX.
2. The driver description text string retuned by …DriverGetDescription contains *version* information, for example "Jenoptik MexCam 3.4.0.24", where "3.4.0.24" represents the *version* number.
3. The value returned by …ExtraGetCount depends on the Jenopik camera. If the value returned is 0 (zero), the Jenopik camera does not have any extra settings available.
4. The description text string returned by …GetDescription contains the *serial number* of the camera and is set by the factory.
5. The value returned by …PortGetCount is always 0 (zero), because Jenoptik cameras have no ancillary analog or digital inputs or outputs.

SIDXDeviceExtra…

"TriggerOutMode"

This setting defines the trigger output mode.

SIDXDeviceExtra…	Returns
…GetName	"TriggerOutMode"
…GetType	SIDX_SETTING_TYPE_LIST
…IsSettable	true
…ListGetCount	4
…ListGetLocal (0)	"no trigger out"
…ListGetLocal (1)	"trigger out during exposure"
…ListGetLocal (2)	"trigger out during readout"
…ListGetLocal (3)	"trigger out during image (including exposure and readout)"

"TriggerOutSignal"

This setting defines the trigger output signal type.

SIDXDeviceExtra…	Returns
…GetName	"TriggerOutSignal"
…GetType	SIDX_SETTING_TYPE_LIST
…IsSettable	true
…ListGetCount	2
…ListGetLocal (0)	"low"
…ListGetLocal (1)	"high"

SIDXCamera…

Specification

SIDXCamera…	Returns
…BinningGetType	SIDX_SETTING_TYPE_LIST
…CoolingControlGet	SIDX_COOLING_CONTROL_NONE
…EMGainGetType	SIDX_SETTING_TYPE_NONE
…ExposeArrayGetSize	0
…ExposeGetRange	0 to *maximum*
…FanControlExists	*camera dependent*
…GainGetType	SIDX_SETTING_TYPE_REAL
…GainGetRange	1.0 to *maximum*
…IntensifierGetType	SIDX_SETTING_TYPE_NONE
…OperationItemGetCount	1 or up to 5
…OperationItemGetLocal (0)	"Default"
…OperationItemGetLocal (1 - 4)	"Microscan *value*, *number* shots"
…ReadoutItemGetCount	2
…ReadoutItemGetLocal	"*pixel_rate*MHz, *pixel_depth* bits"
…TemperatureExists	false

Notes

1. The *maximum* exposure value returned by …ExposeGetRange is camera dependent.
2. The *maximum* gain value returned by …GainGetRange is either 1.0 or 8.0 depending on the camera.
3. Only some cameras support Microscan modes. For Microscan modes the text string returned by …OperateItemGetLocal contain the nominal image size *value* in the name. The *number* of the "shot" is relative to the microscan value. For example, index 1 for …OperateItemGetLocal could return "Microscan 2x2, 2 shots".
4. The *pixel_rate* value and the *pixel_depth* value returned by …ReadoutItemGetLocal depend on the readout item.

SIDXCameraTriggerMode…

Use …TriggerModeExists to obtain the available trigger modes for a specific Jenoptik camera. If SIDX_TRIGGER_IN_MODE_EXPOSURE_DURATION is set, there will be no output signal, regardless of what "TriggerOutMode" item is set.

SIDXTriggerInMode	Available
SIDX_TRIGGER_IN_MODE_ALWAYS	always
SIDX_TRIGGER_IN_MODE_EXPOSURE_DURATION	camera with overlapping readout capability
SIDX_TRIGGER_IN_MODE_SEQUENCE_START	camera dependent

Discussion

Overlapping Readout

Some Jenoptik cameras support reading out the pixel data of the previous image during the exposure of the current image (overlapping readout). This mode is always used when available.

Exposure and Readout Behavior

If the exposure interval is longer than the readout interval, the camera is ready for the next image at the end of the exposure interval of the current image. If the exposure interval is shorter than the readout interval, the exposure interval of the current image does not end until the readout interval of the previous image ends.

2.4 PCO pixelfly

SIDXDevice…

Specification

SIDXDevice…	Returns
…ActionGetCount	0
…DriverGetDescription	"PCO pixelfly pccam *version*"
…DriverGetName	"PCO pixelfly"
…DriverGetType	SIDX_DRIVER_TYPE_COOKE_PCO_PIXELFLY
…ExtraGetCount	0
…GetDescription	"*label text string*"
…GetName	"PCO pixelfly *index*"
…GetLabel	"PCO pixelfly *model*"
…PortGetCount	0

Notes

1. The values returned by …ActionGetCount and …ExtraGetCount are always 0 (zero), because the PCO pixelfly has no device-specific actions or settings.
2. The description text string returned by …GetDescription contains manufacturer hardware information about the PCO pixelfly camera and the PCO pixelfly board.
3. The name text string returned by …GetName contains the PCO pixelfly *index*, where the *index* is a decimal digit in the range 0 to 3, for example "PCO pixelfly 0".
4. The description text string returned by …GetLabel contains the PCO pixelfly *model*, for example "PCO pixelfly qe", where "qe" represents the model. The current possible model names are:
   - "qe"
   - "qe double shutter"
   - "vga"
   - "vga double shutter"
5. The value returned by …PortGetCount is always 0 (zero), because the PCO pixelfly has no ancillary analog or digital inputs or outputs.

SIDXCamera…

Specification

SIDXCamera…	Returns
…BinningGetType	SIDX_SETTING_TYPE_LIST
…CoolingControlGet	SIDX_COOLING_CONTROL_NONE
…EMGainGetType	SIDX_SETTING_TYPE_NONE
…ExposeArrayGetSize	0
…ExposeGetRange	*minimum* to *maximum*
…FanControlExists	false
…GainGetType	SIDX_SETTING_TYPE_LIST
…GainItemGetCount	2
…IntensifierGetType	SIDX_SETTING_TYPE_NONE
…OperationItemGetCount	1 or 2
…OperationItemGetLocal (0)	"Default"
…OperationItemGetLocal (1)	"DoubleShutter"
…ReadoutItemGetCount	1
…ReadoutItemGetLocal (0)	"20MHz, 12 bits"
…ReadoutGetValue	$2x10^7$, 12
…TemperatureExists	true

Notes

1. The *minimum* and *maximum* exposure time returned by …ExposeGetRange is $5x10^{-6}$ to 0.065 seconds if the camera is running to acquire one image in SIDX_TRIGGER_IN_MODE_ALWAYS mode or is running in SIDX_TRIGGER_IN_MODE_EXPOSURE_START mode, otherwise the *minimum* and *maximum* value is 0.001 to 65 seconds.
2. The gain values returned by …GainItemGetEntry are not calibrated gain values, they are nominal gain values. A gain value of 2.0 is twice the gain of a gain value of 1.0, but the absolute gain (conversion of electrons to pixel values) is not specified.
3. The operation mode "DoubleShutter" is only available on a PCO pixelfly double shutter camera which has the double shutter feature enabled.

SIDXCameraExpose…

Use …ExposeGetRange to obtain the exposure time range. The PCO pixelfly supports two ranges of exposure time. For 5µs to 65ms, exposure and readout happen in sequence. The image is completely read out of the camera before exposure of the subsequent image begins. For 1ms to 65s, during the readout of a completed exposure, the next exposure begins. The exposure of the current image and the readout of the previous image overlap in time.

SIDXCameraExposeGetRange	Exposure and Readout Behavior
5µs to 65ms	sequencial
1ms to 65s	overlapped

SIDXCameraTriggerMode…

Use …TriggerModeExists to obtain the available trigger modes. The PCO pixelfly supports up to three modes of trigger input. The trigger mode affects the exposure time range. The exposure time setting is rounded to the nearest 1µs or 1ms, depending on the exposure range.

SIDXTriggerInMode	SIDXCameraExposeGetRange
SIDX_TRIGGER_IN_MODE_ALWAYS	5µs to 65ms for one image 1ms to 65s for more than one image
SIDX_TRIGGER_IN_MODE_EXPOSURE_START	5µs to 65ms
SIDX_TRIGGER_IN_MODE_SEQUENCE_START	1ms to 65s

Discussion

Double Shutter

Some PCO pixelfly cameras are equipped with a double-shutter mode option. If available, an application can select double-shutter mode using the SIDX camera function …OperationSet. The image exposure time range is 5µs to 65ms. Each acquired image will be double the size of a single image, and contain two exposures in sequence. The first exposure will have an exposure time as specified. The second exposure will have an exposure time that is the camera readout time of the first exposure. See the PCO Operating Instructions manual for the specific camera for details.

2.5 Photometrics/Princeton Instruments

SIDXDevice…

Specification

SIDXDevice…	Returns
…ActionGetCount	0
…DriverGetDescription	"PVCAM *version*"
…DriverGetName	"PVCAM"
…DriverGetType	SIDX_DRIVER_TYPE_PVCAM
…ExtraGetCount	*camera dependent*
…GetDescription	*serial number*
…GetName	"PVCAM *name*"
…GetLabel	"PVCAM *model*"
…PortGetCount	0

Notes

1. The driver description text string returned by …DriverGetDescription contains the PVCAM driver ***version***, for example "PVCAM 2.7.21", where "2.7.21" represents the driver***version***.
2. The value retuned by …ExtraGetCount is camera dependent.
3. The ***serial number*** reported by …GetDescription is set by the factory.
4. The value returned by …PortGetCount is always 0 (zero) because the PVCAM has no ancillary analog or digital inputs or outputs.

SIDXDeviceExtra…

"ActiveRows"

This setting enables custom chip feature on the camera if available. The operation redefines the sensor CCD chip dimension so that only the specified number of rows will be readout from the camera.

SIDXDeviceExtra…	Returns
…GetName	"ActiveRows"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	*camera dependent*
…IntegerGetRange	*camera dependent*

"AntiBlooming"

This setting enables/disables anti-blooming feature on the camera if available. It is true to enable anti-blooming.

SIDXDeviceExtra…	Returns
…GetName	"AntiBlooming"
…GetType	SIDX_SETTING_TYPE_BOOLEAN
…IsSettable	true

"CustomChip"

This setting enables/disables the custom chip option if available. It allows the user to change the CCD's dimentions in the software. The ROI setting are based on the custom chip dimensions, not on the actual physical dimensions of the CCD chip. Only after the setting is enabled, PostMaskLine, PostPixelCount, PreMaskLine and PrePixelCount are settable.

SIDXDeviceExtra…	Returns
…GetName	"CustomChip"
…GetType	SIDX_SETTING_TYPE_BOOLEAN
…IsSettable	*camera dependent*

"ADCOffset"

This setting controls the bias offset voltage in the camera. The result is not linear. But a higher offset voltage yields a higher value for all output pixels and a lower offset voltage yields a lower value for all output pixels.

SIDXDeviceExtra…	Returns
…GetName	"ADCOffset"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	*camera dependent*

"ClearCycle"

This setting sets the number of times the CCD must be cleared to remove the charge from the parallel register if available. It should be a positive value.

SIDXDeviceExtra…	Returns
…GetName	"ClearCycle"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	0 to *maximum*

"MinimumBlockCount"

This setting controls the CCD skip parameter on the shift register if available. It is the number of row groups on the shift register and throw away. It should be a positive value.

SIDXDeviceExtra…	**Returns**
…GetName	"MinimumBlockCount"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	0 to *maximum*

"MinimumBlockSize"

This setting controls the CCD skip parameter on the shift register if available. It is the number of rows to group on the shift register and throw away. It should be a positive value and less than the maximum rows on the CCD chip.

SIDXDeviceExtra…	**Returns**
…GetName	"MinimumBlockSize"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	*camera dependent*

"PostMaskLine"

This setting controls the number of masked lines at the far end of the parallel register (away from the serial register). This is the number of additional parallel shifts that need to be done after readout to clear the parallel register. It is settable only if CustomChip is enabled.

SIDXDeviceExtra…	**Returns**
…GetName	"PostMaskLine"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	*camera dependent*

"PostPixelCount"

This setting controls the number of pixels to discard from the serial register after the last real data pixel. These number of pixels must be read or discarded to clear the serial register. It is settable only if CustomChip is enabled.

SIDXDeviceExtra…	**Returns**
…GetName	"PostPixelCount"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	*camera dependent*

"PreMaskLine"

This setting controls the number of masked lines at the near end of the parallel register (next to the serial register). It is settable only if CustomChip is enabled.

SIDXDeviceExtra…	Returns
…GetName	"PreMaskLine"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	*camera dependent*

"PrePixelCount"

This setting controls the number of pixels to discard from the serial register before the first real data pixel. It is settable only if CustomChip is enabled.

SIDXDeviceExtra…	Returns
…GetName	"PrePixelCount"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	*camera dependent*

"SkipAtOnceRowCount"

The setting controls the size of rows to skip at once if available. When the setting is set to none zero, it overwrites the other skip settings. That is, the camera will discard all rows specified at once and ignore any other skip settings. The value should be positive value.

SIDXDeviceExtra…	Returns
…GetName	"SkipAtOnceRowCount"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	*camera dependent*

"StripsPerClear"

The setting controls the number of strips per clear if available. It should be a positive value.

SIDXDeviceExtra…	Returns
…GetName	"StripsPerClear"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	*camera dependent*

"ClearMode"

This setting defines when clearing takes place in the camera if available.

SIDXDeviceExtra…	Returns
…GetName	"ClearMode"
…GetType	SIDX_SETTING_TYPE_LIST
…IsSettable	true
…ListGetCount	6
…ListGetLocal (0)	"clear never: do not perform clear"
…ListGetLocal (1)	"clear pre exposure: perform clear before each exposure"
…ListGetLocal (2)	"clear pre sequence: perform clear before each sequence of images"
…ListGetLocal (3)	"clear post sequence: perform clear after each sequence of images"
…ListGetLocal (4)	"clear pre and post sequence: perform clear before and after each sequence of images"
…ListGetLocal (5)	"clear pre exposure and post sequence: perform clear before the first exposure and after the sequence of images"

"ClockingMode"

The setting controls the parallel clocking method if available.

SIDXDeviceExtra…	Returns
…GetName	"ClockingMode"
…GetType	SIDX_SETTING_TYPE_LIST
…IsSettable	true
…ListGetCount	4
…ListGetLocal (0)	"normal mode"
…ListGetLocal (1)	"frame transfer mode"
…ListGetLocal (2)	"alternate normal mode"
…ListGetLocal (3)	"alternate frame transfer mode"

"IntensifierMode"

The setting controls the parallel clocking method if available. If "IntensifierMode" is set to "Gating" or "Shutter", the instensifier gain can be controlled through SIDXCameraIntensifier… functions.

SIDXDeviceExtra…	**Returns**
…GetName	"IntensifierMode"
…GetType	SIDX_SETTING_TYPE_LIST
…IsSettable	true
…ListGetCount	3
…ListGetLocal (0)	"Off: disabling intensifier"
…ListGetLocal (1)	"Gating: TTL pulse to turn the intensifer on and off"
…ListGetLocal (2)	"Shutter: use the shutter timing to turn the intensifier on and off"

"LogicOutput"

The setting controls the logic output signal from the camera. This is a way for monitoring the camera. Please refer to the camera vendor documentation.

SIDXDeviceExtra…	**Returns**
…GetName	"LogicOutput"
…GetType	SIDX_SETTING_TYPE_LIST
…IsSettable	true
…ListGetCount	*camera dependent*
…ListGetLocal (*index*)	*camera dependent*

"LogicOutputInvert"

The setting controls the polarity of the logic output signal from the camera. When the value is set to true, the logic output signal is low when active. The logic output signal is a way for monitoring the camera. Please refer to the camera vendor documentation.

SIDXDeviceExtra…	**Returns**
…GetName	"LogicOutputInvert"
…GetType	SIDX_SETTING_TYPE_BOOLEAN
…IsSettable	*camera dependent*

"PreAmplifierDelay"

The setting controls the time required for the CCD output pre-amplifier to stablize in second after it is turned on.

SIDXDeviceExtra…	**Returns**
…GetName	"PreAmplifierDelay"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	*camera dependent*

"ShutterCloseDelay"

This setting controls the shutter close delay time in seconds, the time required for the shutter to close. The factory default values compensate for the standard shutter that is shipped with all cameras. You only need to change the setting if you use a shutter with characteristics that differs from the standard shutter.

SIDXDeviceExtra…	**Returns**
…GetName	"ShutterCloseDelay"
…GetType	SIDX_SETTING_TYPE_REAL
…GetUnit	second
…IsSettable	true
…RealGetRange	0 to 65.535

"ShutterOpenDelay"

This setting controls the shutter open delay time in seconds, the time required for the shutter to open. The factory default values compensate for the standard shutter that is shipped with all cameras. You only need to change the setting if you use a shutter with characteristics that differs from the standard shutter. The value range is between zero and 65.535 seconds.

SIDXDeviceExtra…	**Returns**
…GetName	"ShutterOpenDelay"
…GetType	SIDX_SETTING_TYPE_REAL
…GetUnit	second
…IsSettable	true
…RealGetRange	0 to 65.535

"ScriptCommand"

This setting contains the camera commands as string. When the setting is set with non-null string, SIDX will run the camera with the commands and it overwrites all other camera settings. When the setting is set with null string, it clears the script and SIDX will run the camera with the settings set.

SIDXDeviceExtra…	**Returns**
…GetName	"ScriptCommand"
…GetType	SIDX_SETTING_TYPE_STRING
…IsSettable	true
…StringSet	*command script* or *null string*

"Spectra"

This setting defines the exposed bands on the sensor chip as string (starting row and band height). Each band will be vertically binned into a single row by the camera on readout. For example, if the CCD sensor dimension is (1340, 400), you may specify two bands as string in the form of "0, 150, 249, 150". The area on the CCD sensor starting from row 0 and the following 149 rows of pixels will be binned by the hardware into 1 row. The area on the CCD sensor starting from row 249 and the following 149 rows of pixels will be binned by the hardware into 1 row. The CCD sensor dimension may be redefined with "ActiveRows" setting.

SIDXDeviceExtra…	**Returns**
…GetName	"Spectra"
…GetType	SIDX_SETTING_TYPE_STRING
…IsSettable	*camera dependent*
…StringSet	*definition string* or *null string*

"VerticalShift"

The setting controls vertical shift time in seconds of the camera.

SIDXDeviceExtra…	**Returns**
…GetName	"VerticalShift"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	*camera dependent*
…RealGetRange	*camera dependent*

SIDXCamera…

Specification

SIDXCamera…	Returns
…BinningGetType	SIDX_SETTING_TYPE_LIST
…CoolingControlGet	SIDX_COOLING_CONTROL_GET_SET
…EMGainGetType	SIDX_SETTING_TYPE_NONE or SIDX_SETTING_TYPE_LIST
…ExposeArrayGetSize	0
…ExposeGetRange	0 to *maximum*
…FanControlExists	false
…GainGetType	SIDX_SETTING_TYPE_LIST or SIDX_SETTING_TYPE_REAL
…GainItemGetCount	*camera dependent*
…IntensifierGetType	SIDX_SETTING_TYPE_REAL
…IntensifierGetRange	0 to 255
…OperateItemGetCount	*camera dependent*
…ReadoutItemGetCount	1 or camera and operation mode dependent
…TemperatureExists	true

Notes

1. The *maximum* value returned by …ExposeGetRange is camera dependent.
2. The value returned by …GainGetType is either SIDX_SETTING_TYPE_LIST, where gain is a list of possible values, or SIDX_SETTING_TYPE_REAL, where gain is a range of possible values. The return type is camera dependent.
3. …Intensifier… operations only have an effect when the SIDXDeviceExtraList… setting "IntensifierMode" is not set to "Off".
4. The available Operate modes are camera dependent.

SIDXCameraShutterMode…

Use …ShutterModeExists to obtain the available shutter modes for a specific PVCAM camera.

SIDXShutterMode	Available
SIDX_SHUTTER_MODE_OPEN_DURING_EXPOSURE	always
SIDX_SHUTTER_MODE_OPEN	always
SIDX_SHUTTER_MODE_CLOSE	always

SIDXCameraTriggerMode…

Use …TriggerModeExists to obtain the available trigger input modes for a specific PVCAM camera.

SIDXTriggerInMode	Available
SIDX_TRIGGER_IN_MODE_ALWAYS	always
SIDX_TRIGGER_IN_MODE_EXPOSURE_START	always
SIDX_TRIGGER_IN_MODE_EXPOSURE_DURATION	camera dependent
SIDX_TRIGGER_IN_MODE_SEQUENCE_START	always

Discussion

Kinetic Mode

This is a mode where there are multiple exposures and one readout. The maximum number of exposures per readout is CCD height/kinetic window height. The camera runs in "burst mode". The exposures within one readout are acquired uniformly in time. SIDX runs the camera in this mode if the camera is capable and the requested images fit in one readout. For example, if the camera is capable to run in kinetic mode, the ROI height is 200 rows and CCD height is 1000 rows, the acquire image limit is set to 5, SIDX will run the camera in kinetic mode.

2.6 SciMeasure Analytical Systems

SIDXDevice…

Specification

SIDXDevice…	Returns
…ActionGetCount	0
…DriverGetDescription	"SciMeasure Analytical System LittleJoe *version*"
…DriverGetName	"SciMeasure Analytical System LittleJoe"
…DriverGetType	SIDX_DRIVER_TYPE_SCIMEASURE
…ExtraGetCount	5
…GetDescription	"*baud rate*"
…GetName	"SciMeasure Analytical System *index*"
…GetLabel	"SciMeasure Analytical System *model*"
…PortGetCount	0

Notes

1. The values returned by …ActionGetCountis always 0 (zero), because the SciMeasure Ananlytical System has no device-specific actions.
2. The description text string returned by …GetDescription contains manufacturer hardware information about the SciMeasure Ananlytical System camera.
3. The name text string returned by …GetName contains the SciMeasure Ananlytical System *index*, where the *index* is a decimal digit in the range greater than or equal to 0, for example "SciMeasure Ananlytical System 0".
4. The description text string returned by …GetLabel contains the SciMeasure Ananlytical System *model*, for example "SciMeasure Ananlytical System CCD39".
5. The value returned by …PortGetCount is always 0 (zero), because the SciMeasure Ananlytical System has no ancillary analog or digital inputs or outputs.

SIDXDeviceExtra…

"AccumulationCount"

This setting is an integer value in the range between 1 and 65536. The default value is set at factory. The setting sets the number of times the accumulation segment is repeated

SIDXDeviceExtra…	Returns
…GetName	"AccumulationCount"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…RealGetRange	1 to 65536

"ChannelOffset"

This setting is a string which has an array of interger values in the range 0 to 1023 separated by space. The maximum entries in the string is 16. Each entry represents the offset of one channel. For example, the first entry in the string is the offset for channel 0 (zero, the first channel). The default values are set at the factory. The setting controls the black level of the image. The greater the value, the darker the image.

SIDXDeviceExtra…	**Returns**
…GetName	"ChannelOffset"
…GetType	SIDX_SETTING_TYPE_STRING
…IsSettable	true
…StringGet	*offset offset … (up to 16)*

"DelayClampSignal"

Set the delay for the clamp signal in seconds for all channels. The setting is a string which has an array of real values separated by space. Each entry represents the delay of one channel. The setting adjusts the timing of the sample and clamp signals for the channels. The resolution of the setting is 0.25 ns.

SIDXDeviceExtra…	**Returns**
…GetName	"DelayClampSignal"
…GetType	SIDX_SETTING_TYPE_STRING
…IsSettable	true
…StringGet	*delay delay …*

"ImageInterval"

This is a setting for SIDX (not the camera). It is a real value in seconds. If the actual interval between images is less than the value, SIDX will collapse multiple images into one before reading out in order to catch up with the fast frame rate.

SIDXDeviceExtra…	**Returns**
…GetName	"ImageInterval"
…GetType	SIDX_SETTING_TYPE_REAL
…GetUnit	second
…IsSettable	true
…RealGetRange	*available at runtime*

"ImageRaw"

This is a boolean value. It is true for providing the raw image pixels to the application. Otherwise not. The default value is true.

SIDXDeviceExtra…	**Returns**
…GetName	"ImageRaw"
…GetType	SIDX_SETTING_TYPE_BOOLEAN
…IsSettable	true

"PixelInversion"

This is a boolean value. It is true for inverting each pixel value before transfering it to the host computer. Otherwise not. This is a hardware operation and has no impact on performance.

SIDXDeviceExtra…	**Returns**
…GetName	"PixelInversion"
…GetType	SIDX_SETTING_TYPE_BOOLEAN
…IsSettable	true

SIDXCamera…

Specification

SIDXCamera…	**Returns**
…BinningGetType	SIDX_SETTING_TYPE_LIST
…CoolingControlGet	SIDX_COOLING_CONTROL_GET_ONLY
…EMGainGetType	SIDX_SETTING_TYPE_NONE
…ExposeArrayGetSize	0
…ExposeGetRange	*minimum* to *maximum*
…FanControlExists	false
…GainGetType	SIDX_SETTING_TYPE_LIST
…GainItemGetCount	4
…IntensifierGetType	SIDX_SETTING_TYPE_NONE
…OperateItemGetCount	*camera dependent*
…OperateItemGetLocal (0-7)	"Program *number*, *pixel_rate*MHz, *width_pixel* x *height_pixel*"
…ReadoutItemGetCount	1
…ReadoutItemGetLocal	"*pixel_rate*MHz, *pixel_depth* bits"
…ReadoutGetValue	*pixel_rate, pixel_depth*
…TemperatureExists	true

Notes

1. The *minimum* and *maximum* exposure values returned by …ExposeGetRange are camera dependent.
2. The gain values returned by …GainItemGetEntry are not calibrated gain values, they are nominal gain values. A gain value of 2.0 is the higher gain value than a gain value of 1.0, but the absolute gain (conversion of electrons to pixel values) is not specified.
3. The text string returned by …OperateItemGetLocal contains the *pixel_rate* and image *width_pixel* by *height_pixel* that corresponds to the operation mode. For example, index 3 for …OperateItemGetLocal could return "Program 3, 1000Hz, 80x80".
4. The *pixel_rate* value and the *pixel_depth* value returned by …ReadoutGetValue depend on the operation mode.

SIDXCameraTriggerMode...

SIDXTriggerInMode	Available
SIDX_TRIGGER_IN_MODE_ALWAYS	Always
SIDX_TRIGGER_IN_MODE_EXPOSURE_START	Always
SIDX_TRIGGER_IN_MODE_EXPOSURE_DURATION	Always

Discussion

High Speed Imaging

When recording images at high speed, SIDX operates the SciMeasure camera in a mode such that images must occur in groups. An image group will be a small power of two images, for example, 1, 2, 4, 8, or 16 images. SIDX acquires entire image groups. For example, if SIDX uses groups of 8 images, and SIDX acquires 12 images, the camera will actually record 16 images, of which SIDX will record only 12. As already mentioned, this mode is used only when acquiring at very high rates.

High Speed Triggering

When recording images at high speed, this recording mode affects the number of trigger input and output pulses. For example, if an application requests 12 images from SIDX using external triggering, and SIDX chooses to acquire two groups of 8 images, the camera must receive 16 triggers, one for each image, before SIDX will acquire all 12 images. Similarly, when using internal triggering, SIDX would produce 16 triggers when acquiring 12 images.

3. Motorized Stage

3.1 ASI

SIDXDevice…

Specification

SIDXDevice…	Returns
…ActionGetCount	13
…DriverGetDescription	
…DriverGetName	Applied Scientific Instrumentation
…DriverGetType	SIDX_DRIVER_TYPE_APPLIED_SCIENTIFIC_INSTRUMENTATION
…ExtraGetCount	23
…GetDescription	Compile date: *date* Build information: *motor*
…GetName	Applied Scientific Instrumentation *model*
…GetLabel	ASI-*model*
…PortGetCount	6

Notes
1. The value returned by …DriverGetDescription is an empty string.
2. The value returned by …PortGetCount combines the count of both analog and digital I/O ports.

SIDXDeviceAction…

"AutoAlignXY"

Automatically align the stage.

"AutoAlignZ"

Automatically align the focus.

"AutoZeroXY"

Automatically re-zero the stage.

"AutoZeroZ"

Automatically re-zero the focus.

"ClearTrajectoryBuffer"

Clears the trajectory buffer.

"DumpErrorBuffer"

Dumps the error buffer.

"DumpTrajectoryBuffer"

Dumps the trajectory buffer.

"LCDUnits"

Toggle the units on the stage screen.

"SaveRestoreSettings"

Restore previous settings.

"SaveReloadFactorySettings"

Restore factory defaults.

"SaveSettingsToFlash"

Save settings to flash.

"ScanRaster"

Start the scan using the settings from the last call to ScanRasterSettings. This function requires special SCAN firmware on the ASI stage.

SIDXDeviceExtra…

"ButtonEnable"

Enables and disables all buttons on the stage.

SIDXDeviceExtra…	**Returns**
…GetName	"ButtonEnable"
…GetType	SIDX_SETTING_TYPE_BOOLEAN
…IsSettable	true
…BooleanGet	*enable all*

"ButtonSettings"

Allows the user to change button settings on the stage.

SIDXDeviceExtra…	**Returns**
…GetName	"ButtonSettings"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	*0 - 255*

"DriftErrorXY"

Set the drift error before the stage repositions. The drift error is in units of microns.

SIDXDeviceExtra…	Returns
…GetName	"DriftErrorXY"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	*0 - 10000*

"DriftErrorZ"

Set the drift error before the focus repositions. The drift error is in units of microns.

SIDXDeviceExtra…	Returns
…GetName	"DriftErrorZ"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	*0 - 10000*

"LCDText"

Set the text on the stage screen, any '~' characters will be replaced with '-' to prevent stage resets.

SIDXDeviceExtra…	Returns
…GetName	"LCDText"
…GetType	SIDX_SETTING_TYPE_STRING
…IsSettable	true

"LimitPositionLowGetX"

Get the lower limits of X axis motion for the stage in meters. The values were retrieved during that last successful LimitPositionQuery.

SIDXDeviceExtra…	Returns
…GetName	"LimitPositionLowGetX"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	-2.0 - 2.0

"LimitPositionLowGetY"

Get the lower limits of Y axis motion for the stage in meters. The values were retrieved during that last successful LimitPositionQuery.

SIDXDeviceExtra…	Returns
…GetName	"LimitPositionLowGetY"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	-2.0 - 2.0

"LimitPositionLowGetZ"

Get the lower limits of Z axis motion for the stage in meters. The values were retrieved during that last successful LimitPositionQuery.

SIDXDeviceExtra…	Returns
…GetName	"LimitPositionLowGetZ"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	-2.0 - 2.0

"LimitPositionLowSetX"

Set the lower limits of motion for the X axis on the stage in meters.

SIDXDeviceExtra…	Returns
…GetName	"LimitPositionLowSetX"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	-2.0 - 2.0

"LimitPositionLowSetY"

Set the lower limits of motion for the Y axis on the stage in meters.

SIDXDeviceExtra…	Returns
…GetName	"LimitPositionLowSetY"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	-2.0 - 2.0

"LimitPositionLowSetZ"

Set the lower limits of motion for the Z axis on the stage in meters.

SIDXDeviceExtra…	Returns
…GetName	"LimitPositionLowSetY"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	-2.0 - 2.0

"LimitPositionQuery"

Query the stage for the limit positions. The positions can be retrieved with the get functions.

"LimitPositionUpGetX"

Get the upper limits of X axis motion for the stage in meters. The values were retrieved during that last successful LimitPositionQuery.

SIDXDeviceExtra…	Returns
…GetName	"LimitPositionLowSetY"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	-2.0 - 2.0

"LimitPositionUpGetY"

Get the upper limits of Y axis motion for the stage in meters. The values were retrieved during that last successful LimitPositionQuery.

SIDXDeviceExtra…	Returns
…GetName	"LimitPositionUpGetY"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	-2.0 - 2.0

"LimitPositionUpGetZ"

Get the upper limits of Z axis motion for the stage in meters. The values were retrieved during that last successful LimitPositionQuery.

SIDXDeviceExtra…	Returns
…GetName	"LimitPositionUpGetZ"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	-2.0 - 2.0

"LimitPositionUpSetX"

Set the upper limits of motion for the X axis on the stage in meters.

SIDXDeviceExtra…	Returns
…GetName	"LimitPositionUpSetX"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	-2.0 - 2.0

"LimitPositionUpSetY"

Set the upper limits of motion for the Y axis on the stage in meters.

SIDXDeviceExtra…	Returns
…GetName	"LimitPositionUpSetY"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	-2.0 - 2.0

"LimitPositionUpSetZ"

Set the upper limits of motion for the Z axis on the stage in meters.

SIDXDeviceExtra…	Returns
…GetName	"LimitPositionUpSetZ"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	-2.0 - 2.0

"Maintain"

Set the behavior after arriving on position. The code for the axes is an integer array.

SIDXDeviceExtra…	Returns
…GetName	"Maintain"
…GetType	SIDX_SETTING_TYPE_SEQUENCE
…IsSettable	true
…SequenceGetSize	4
…SequenceGet	[0] *x axis* [1] *y axis* [2] *z axis* [3] *f axis*

"ResolutionSetZ"

Set the resolution of the focus in micrometers.

SIDXDeviceExtra…	Returns
…GetName	"ResolutionSetZ"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	0 - 10000

"SavePosition"

Set the behavior on saving positions on power off.

SIDXDeviceExtra…	Returns
…GetName	"SavePosition"
…GetType	SIDX_SETTING_TYPE_BOOLEAN
…IsSettable	true
…BooleanGet	*save on power off*

"ScanRasterSettings"

Set the behavior of the raster scan. The array values are of type integer.

SIDXDeviceExtra…	Returns
…GetName	"ScanRasterSettings"
…GetType	SIDX_SETTING_TYPE_SEQUENCE
…IsSettable	true
…SequenceGetSize	12
…SequenceGet	[0] [1] [2] [3] *SCAN settings (unitless)* [4] [5] *SCANR settings (nanometers)* [6] [7] *SCANR settings (unitless)* [8] [9] *SCANV settings (nanometers)* [10] [11] *SCANV settings (unitless)*

"ServoSetGains"

Set the servo loop error limit in millimeters.

SIDXDeviceExtra…	Returns
…GetName	"ServoSetGains"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	1 - 1000

"TriggerPositionX"

Set the trigger x position in meters.

SIDXDeviceExtra…	Returns
…GetName	"TriggerPositionX"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true

"TriggerPositionY"

Set the trigger y position in meters.

SIDXDeviceExtra…	Returns
…GetName	"TriggerPositionY"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true

"TriggerActive"

Sets the mode of trigger/digital output. Since the digital output and trigger share one pin, the stage can only do one at a time.

SIDXDeviceExtra…	Returns
…GetName	"TriggerActive"
…GetType	SIDX_SETTING_TYPE_BOOLEAN
…IsSettable	true
…BooleanGet	*true* activate trigger, deactivate digital output *false* deactivate trigger, activate digital output

"TriggerSettings"

Sets the trigger input and output modes and settings, the time delay and pulse length. The array values are of type integer.

SIDXDeviceExtra…	Returns
…GetName	"TriggerSettings"
…GetType	SIDX_SETTING_TYPE_SEQUENCE
…IsSettable	true
…SequenceGetSize	6
…SequenceGet	[0] *input mode* [1] *output mode* [2] *auxiliary IO mode* [3] *output polarity* [4] *time delay* [5] *pulse length*

"TriggerZ"

Starts the Z trigger, which moves the Z axis when a high edge is detected. The array values are of type integer.

SIDXDeviceExtra…	Returns
…GetName	"TriggerZ"
…GetType	SIDX_SETTING_TYPE_SEQUENCE
…IsSettable	true
…SequenceGetSize	4
…SequenceGet	[0] *dz increment distance* [1] *num repeats* [2] *mode* [3] *timeout*

"WaitAfterMoveXY"

Set the time to pause after a move is complete. This time is set in milliseconds. The default time is 0 milliseconds.

SIDXDeviceExtra…	Returns
…GetName	"WaitAfterMoveXY"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	0 - 100000

"WaitAfterMoveZ"

Set the time to pause after a move is complete. This time is set in milliseconds. The default time is 0 milliseconds.

SIDXDeviceExtra…	Returns
…GetName	"WaitAfterMoveZ"
…GetType	SIDX_SETTING_TYPE_INTEGER
…IsSettable	true
…IntegerGetRange	0 - 100000

3.2 Prior Scientific

SIDXDevice…

Specification

SIDXDevice…	Returns
…ActionGetCount	2
…DriverGetDescription	
…DriverGetName	Prior Scientific
…DriverGetType	SIDX_DRIVER_TYPE_PRIOR_SCIENTIFIC
…ExtraGetCount	6
…GetDescription	Serial number: *model* Information: *controller peripherals*
…GetName	Prior Scientific *model*
…GetLabel	*model identification*
…PortGetCount	8

Notes

1. The value returned by …DriverGetDescription is an empty string.
2. The value returned by …PortGetCount combines the count of both analog and digital I/O ports.

SIDXDeviceAction…

"ResetIndex"

Set the hardware zero, typically only run ONCE after manufacturing.

"RestoreIndex"

Resynchronizes the stage after manual movement.

SIDXDeviceExtra…

"LoaderForceSlideAsFitted"

SIDXDeviceExtra…	Returns
…GetName	"LoaderForceSlideAsFitted"
…GetType	SIDX_SETTING_TYPE_SEQUENCE
…IsSettable	true
…SequenceGetSize	2
…SequenceGet	[0] *position_cassette* [1] *position_slot*

"LoaderSingleStepDebug"

SIDXDeviceExtra…	Returns
…GetName	"LoaderSingleStepDebug"
…GetType	SIDX_SETTING_TYPE_BOOLEAN
…IsSettable	true
…BooleanGet	*debug*

"LoaderSlideTransfer"

SIDXDeviceExtra…	Returns
…GetName	"LoaderSlideTransfer"
…GetType	SIDX_SETTING_TYPE_SEQUENCE
…IsSettable	true
…SequenceGetSize	4
…SequenceGet	[0] *position_cassette_1* [1] *position_slot_1* [2] *position_cassette_2* [3] *position_slot_2*

"PortBitConfigure"

Configures the bit triggering mechanism. The array values are of type integer.

SIDXDeviceExtra…	Returns
…GetName	"PortBitConfigure"
…GetType	SIDX_SETTING_TYPE_SEQUENCE
…IsSettable	true
…SequenceGetSize	3
…SequenceGet	[0] *enabled (0 true or 1 false)* [1] *port_index* [2] *assert_on_motion (0 true or 1 false)*

"SCurveSetXY"

Sets the s-curve fraction for acceleration profile.

SIDXDeviceExtra	Returns
…GetName	"SCurveSetXY"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	*0.0 - 1.0*

"SCurveSetZ"

Sets the s-curve fraction for acceleration profile.

SIDXDeviceExtra	Returns
…GetName	"SCurveSetZ"
…GetType	SIDX_SETTING_TYPE_REAL
…IsSettable	true
…RealGetRange	*0.0 - 1.0*

4. ITT IDL API

4.1 SIDXRoot

An object of this interface represents the entire SIDX interface. An application (process) should have at most one SIDX object.

Reference

status = **SIDXRootArchiveOpen**(*SIDXRoot*, *path*, *SIDXArchive*)
status = **SIDXRootCameraOpenName**(*SIDXRoot*, *name*, *SIDXCamera*)
status = **SIDXRootCameraScan**(*SIDXRoot*)
status = **SIDXRootCameraScanGetCount**(*SIDXRoot*, *count*)
status = **SIDXRootCameraScanGetLabel**(*SIDXRoot*, *index*, *label*)
status = **SIDXRootCameraScanGetName**(*SIDXRoot*, *index*, *name*)
status = **SIDXRootCameraScanGetReport**(*SIDXRoot*, *report*)
status = **SIDXRootClose**(*SIDXRoot*)
SIDXRootGetLastError(*SIDXRoot*, *message*)
SIDXRootOpen(*SIDXRoot*, *license*)
status = **SIDXRootEnableCapabilities**(*SIDXRoot*, *capability*)
status = **SIDXRootSoftwareGetDescription**(*SIDXRoot*, *description*)
status = **SIDXRootSoftwareGetLicense**(*SIDXRoot*, *license*)
status = **SIDXRootSoftwareGetSerial**(*SIDXRoot*, *serial*)
status = **SIDXRootSoftwareIsLicensed**(*SIDXRoot*, *licensed*)
status = **SIDXRootSoftwareSetLicense**(*SIDXRoot*, *license*)
status = **SIDXRootStageOpen**(*SIDXRoot*, *name*, *port*, *SIDXStage*)

Methods

SIDXRootArchiveOpen

status = **SIDXRootArchiveOpen**(*SIDXRoot*, *path*, *SIDXArchive*)

Parameters
- *SIDXRoot* A handle value that references the SIDXRoot context
- *path* A text string containing the file system path for the archive file to create.
- *SIDXArchive* A variable to receive an object to use to access the image archive.

Return value
- *status* Non-zero if the function failed, zero if it succeeded.

Open a file as an image archive.

SIDXRootCameraOpenName

status = **SIDXRootCameraOpenName**(*SIDXRoot*, *name*, *SIDXCamera*)

Parameters

- *SIDXRoot* A handle value that references the SIDXRoot context
- *name* A text string containing the name of the camera. If the string is empty, SIDX will open the one and only camera connected to the system. If there is more than one camera connected to the system, an empty string will return an error.
- *SIDXCamera* A variable to receive an object to use for access to the camera.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Connect to the specified camera by name. The camera name string can be only a driver name, for example, "PCO pixelfly" or "Andor Technology". In the case that a driver owns multiple cameras, the camera name string must include the the index value.

SIDXRootCameraScan

status = **SIDXRootCameraScan**(*SIDXRoot*)

Parameters

- *SIDXRoot* A handle value that references the SIDXRoot context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Scan cameras. No cameras can be open when enumeration occurs.

SIDXRootCameraScanGetCount

status = **SIDXRootCameraScanGetCount**(*SIDXRoot*, *count*)

Parameters

- *SIDXRoot* A handle value that references the SIDXRoot context
- *count* A variable to receive an integer count of connected cameras, including any simulated cameras.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

This operation scans the computer system to locate any available cameras. The operation cannot be performed if SIDX has any cameras open. The number of available cameras may be zero or more. Use CameraScanGetCount to obtain the number of cameras found on the computer system. This value may be zero. The operation creates a text report that may be useful if unexpected results occur. Use CameraScanGetReport to obtain this report.

SIDXRootCameraScanGetLabel

status = **SIDXRootCameraScanGetLabel**(*SIDXRoot*, *index*, *label*)

Parameters

- *SIDXRoot* A handle value that references the SIDXRoot context
- *index* The integer index of the camera.
- *label* A variable to receive a text string label of the camera.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the label representing the camera. The label is intended to be human readable.

SIDXRootCameraScanGetName

status = **SIDXRootCameraScanGetName**(*SIDXRoot*, *index*, *name*)

Parameters

- *SIDXRoot* A handle value that references the SIDXRoot context
- *index* The integer index of the camera.
- *name* A variable to receive a text string name of the camera.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the name of the camera corresponding to the index. This name is used to open the camera.

SIDXRootCameraScanGetReport

status = **SIDXRootCameraScanGetReport**(*SIDXRoot*, *report*)

Parameters

- *SIDXRoot* A handle value that references the SIDXRoot context
- *report* A variable to receive a text string report corresponding to the most recent camera scan.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a detailed report of all available cameras on the system.

SIDXRootClose

status = **SIDXRootClose**(*SIDXRoot*)

Parameters

- *SIDXRoot* A handle value that references the SIDXRoot context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Close SIDX. This should be the last call to SIDX before the object is discarded.

SIDXRootGetLastError

SIDXRootGetLastError(*SIDXRoot*, *message*)

Parameters

- *SIDXRoot* A handle value that references the SIDX root context.
- *message* A text string to receive the text associated with the error code specified.

The function translates the error code into a text string.

SIDXRootOpen

SIDXRootOpen(*SIDXRoot*, *license*)

Parameters

- *SIDXRoot* A handle value that references the SIDX root context.
- *license* A text string containing a valid SIDX license. If the string is null or invalid, SIDX will look for the license file.

The function returns a handle used to open SIDX.

SIDXRootEnableCapabilities

status = **SIDXRootEnableCapabilities**(*SIDXRoot*, *capability*)

Parameters

- *SIDXRoot* A handle value that references the SIDXRoot context
- *capability* the capability to enable.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Enable a special camera capability. The supported capabilities are:

1000: Enable Photometrics PVCAM support

1001: Enable Princeton Instruments PVCAM support

The function is only supported in WaveMetric IGOR Pro interface.

SIDXRootSoftwareGetDescription

status = **SIDXRootSoftwareGetDescription**(*SIDXRoot*, *description*)

Parameters

- *SIDXRoot* A handle value that references the SIDXRoot context
- *description* A variable to receive a text string software description.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the software description as a text string. The software description includes the product and version, for example: 'SIDX 7.0.1'.

SIDXRootSoftwareGetLicense

status = **SIDXRootSoftwareGetLicense**(*SIDXRoot*, *license*)

Parameters

- *SIDXRoot* A handle value that references the SIDXRoot context
- *license* A variable to receive a text string software license code.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the software license string.

SIDXRootSoftwareGetSerial

status = **SIDXRootSoftwareGetSerial**(*SIDXRoot*, *serial*)

Parameters

- *SIDXRoot* A handle value that references the SIDXRoot context
- *serial* A variable to receive a text string software serial number.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the software serial number as a text string.

SIDXRootSoftwareIsLicensed

status = **SIDXRootSoftwareIsLicensed**(*SIDXRoot*, *licensed*)

Parameters

- *SIDXRoot* A handle value that references the SIDXRoot context
- *licensed* A variable to receive a boolean value, true if the software is licensed, otherwise false.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Verify if the SIDX software license is valid. If the software is not licensed all SIDX calls will return an invalid operation status code.

SIDXRootSoftwareSetLicense

status = **SIDXRootSoftwareSetLicense**(*SIDXRoot*, *license*)

Parameters

- *SIDXRoot* A handle value that references the SIDXRoot context
- *license* A text string software license code.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets an SIDX license. If there is an existing license it will be overwritten.

SIDXRootStageOpen

status = **SIDXRootStageOpen**(*SIDXRoot*, *name*, *port*, *SIDXStage*)

Parameters

- *SIDXRoot* A handle value that references the SIDXRoot context
- *name* A text string containing the name of the stage. An empty string will return an error.
- *port* A string specifying the serial port to be used. An empty string will return an error.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Open the stage controller, by name, on the specified port.

4.2 SIDXCamera

An object of this interface represents an open camera in SIDX.

Reference

status = **SIDXCameraAcquireImageGetLimit**(*SIDXCamera*, *maximum*)
status = **SIDXCameraAcquireImageSetLimit**(*SIDXCamera*, *maximum*)
status = **SIDXCameraAcquireOpen**(*SIDXCamera*, *SIDXAcquire*)
status = **SIDXCameraBinningGet**(*SIDXCamera*, *x*, *y*)
status = **SIDXCameraBinningGetType**(*SIDXCamera*, *type*)
status = **SIDXCameraBinningItemGet**(*SIDXCamera*, *item*)
status = **SIDXCameraBinningItemGetCount**(*SIDXCamera*, *count*)
status = **SIDXCameraBinningItemGetEntry**(*SIDXCamera*, *item*, *x*, *y*)
status = **SIDXCameraBinningItemGetLocal**(*SIDXCamera*, *item*, *description*)
status = **SIDXCameraBinningItemSet**(*SIDXCamera*, *item*)
status = **SIDXCameraBinningSet**(*SIDXCamera*, *x*, *y*)
status = **SIDXCameraBinningXGetLimit**(*SIDXCamera*, *maximum*)
status = **SIDXCameraBinningXGetType**(*SIDXCamera*, *type*)
status = **SIDXCameraBinningXItemGetCount**(*SIDXCamera*, *count*)
status = **SIDXCameraBinningXItemGetEntry**(*SIDXCamera*, *item*, *binning*)
status = **SIDXCameraBinningXItemGetLocal**(*SIDXCamera*, *item*, *description*)
status = **SIDXCameraBinningYGetLimit**(*SIDXCamera*, *maximum*)
status = **SIDXCameraBufferCountGet**(*SIDXCamera*, *count*)
status = **SIDXCameraBufferCountSet**(*SIDXCamera*, *count*)
status = **SIDXCameraClose**(*SIDXCamera*)
status = **SIDXCameraCoolingGet**(*SIDXCamera*, *temperature*)
status = **SIDXCameraCoolingGetControl**(*SIDXCamera*, *control*)
status = **SIDXCameraCoolingGetRange**(*SIDXCamera*, *minimum*, *maximum*)
status = **SIDXCameraCoolingGetValue**(*SIDXCamera*, *temperature*)
status = **SIDXCameraCoolingSet**(*SIDXCamera*, *temperature*)
status = **SIDXCameraEMGainGet**(*SIDXCamera*, *gain*)
status = **SIDXCameraEMGainGetLabel**(*SIDXCamera*, *label*)
status = **SIDXCameraEMGainGetRange**(*SIDXCamera*, *minimum*, *maximum*)
status = **SIDXCameraEMGainGetType**(*SIDXCamera*, *type*)
status = **SIDXCameraEMGainGetUnit**(*SIDXCamera*, *unit*)
status = **SIDXCameraEMGainGetValue**(*SIDXCamera*, *gain*)
status = **SIDXCameraEMGainItemGet**(*SIDXCamera*, *item*)
status = **SIDXCameraEMGainItemGetCount**(*SIDXCamera*, *count*)
status = **SIDXCameraEMGainItemGetEntry**(*SIDXCamera*, *item*, *gain*)
status = **SIDXCameraEMGainItemGetLocal**(*SIDXCamera*, *item*, *description*)
status = **SIDXCameraEMGainItemSet**(*SIDXCamera*, *item*)
status = **SIDXCameraEMGainSet**(*SIDXCamera*, *gain*)

status = **SIDXCameraExposeArrayGet**(*SIDXCamera*, *duration*)
status = **SIDXCameraExposeArrayGetSize**(*SIDXCamera*, *value*)
status = **SIDXCameraExposeArrayGetValue**(*SIDXCamera*, *duration*)
status = **SIDXCameraExposeArraySet**(*SIDXCamera*, *duration*)
status = **SIDXCameraExposeGet**(*SIDXCamera*, *duration*)
status = **SIDXCameraExposeGetRange**(*SIDXCamera*, *minimum*, *maximum*)
status = **SIDXCameraExposeGetValue**(*SIDXCamera*, *duration*)
status = **SIDXCameraExposeSet**(*SIDXCamera*, *duration*)
status = **SIDXCameraExternalDelayGet**(*SIDXCamera*, *interval*)
status = **SIDXCameraExternalDelaySet**(*SIDXCamera*, *interval*)
status = **SIDXCameraFanControlExists**(*SIDXCamera*, *available*)
status = **SIDXCameraFanControlGet**(*SIDXCamera*, *enable*)
status = **SIDXCameraFanControlSet**(*SIDXCamera*, *enable*)
status = **SIDXCameraGainGet**(*SIDXCamera*, *gain*)
status = **SIDXCameraGainGetLabel**(*SIDXCamera*, *label*)
status = **SIDXCameraGainGetRange**(*SIDXCamera*, *minimum*, *maximum*)
status = **SIDXCameraGainGetType**(*SIDXCamera*, *type*)
status = **SIDXCameraGainGetUnit**(*SIDXCamera*, *unit*)
status = **SIDXCameraGainGetValue**(*SIDXCamera*, *gain*)
status = **SIDXCameraGainItemGet**(*SIDXCamera*, *item*)
status = **SIDXCameraGainItemGetCount**(*SIDXCamera*, *count*)
status = **SIDXCameraGainItemGetEntry**(*SIDXCamera*, *item*, *gain*)
status = **SIDXCameraGainItemGetLocal**(*SIDXCamera*, *item*, *description*)
status = **SIDXCameraGainItemSet**(*SIDXCamera*, *item*)
status = **SIDXCameraGainSet**(*SIDXCamera*, *gain*)
SIDXCameraGetLastError(*SIDXCamera*, *message*)
status = **SIDXCameraIntensifierGet**(*SIDXCamera*, *gain*)
status = **SIDXCameraIntensifierGetLabel**(*SIDXCamera*, *label*)
status = **SIDXCameraIntensifierGetRange**(*SIDXCamera*, *minimum*, *maximum*)
status = **SIDXCameraIntensifierGetType**(*SIDXCamera*, *type*)
status = **SIDXCameraIntensifierGetUnit**(*SIDXCamera*, *unit*)
status = **SIDXCameraIntensifierGetValue**(*SIDXCamera*, *gain*)
status = **SIDXCameraIntensifierSet**(*SIDXCamera*, *gain*)
status = **SIDXCameraOperateGet**(*SIDXCamera*, *value*)
status = **SIDXCameraOperateItemGet**(*SIDXCamera*, *item*)
status = **SIDXCameraOperateItemGetCount**(*SIDXCamera*, *count*)
status = **SIDXCameraOperateItemGetLocal**(*SIDXCamera*, *item*, *description*)
status = **SIDXCameraOperateItemSet**(*SIDXCamera*, *item*)
status = **SIDXCameraOperateSet**(*SIDXCamera*, *value*)
status = **SIDXCameraPollingGet**(*SIDXCamera*, *interval*)
status = **SIDXCameraPollingSet**(*SIDXCamera*, *interval*)
status = **SIDXCameraReadoutGet**(*SIDXCamera*, *value*)
status = **SIDXCameraReadoutGetValue**(*SIDXCamera*, *pixel_rate*, *pixel_depth*)
status = **SIDXCameraReadoutItemGet**(*SIDXCamera*, *item*)
status = **SIDXCameraReadoutItemGetCount**(*SIDXCamera*, *count*)

status = **SIDXCameraReadoutItemGetEntry**(*SIDXCamera*, *item*, *pixel_rate*, *pixel_depth*)
status = **SIDXCameraReadoutItemGetLocal**(*SIDXCamera*, *item*, *description*)
status = **SIDXCameraReadoutItemSet**(*SIDXCamera*, *item*)
status = **SIDXCameraReadoutSet**(*SIDXCamera*, *value*)
status = **SIDXCameraROIClear**(*SIDXCamera*)
status = **SIDXCameraROIGet**(*SIDXCamera*, *x1*, *y1*, *x2*, *y2*)
status = **SIDXCameraROIGetValue**(*SIDXCamera*, *x1*, *y1*, *x2*, *y2*)
status = **SIDXCameraROISet**(*SIDXCamera*, *x1*, *y1*, *x2*, *y2*)
status = **SIDXCameraRotateClear**(*SIDXCamera*)
status = **SIDXCameraRotateMirrorX**(*SIDXCamera*)
status = **SIDXCameraRotateMirrorY**(*SIDXCamera*)
status = **SIDXCameraRotateSet**(*SIDXCamera*, *count*)
status = **SIDXCameraShutterExists**(*SIDXCamera*, *mode*, *available*)
status = **SIDXCameraShutterGet**(*SIDXCamera*, *mode*)
status = **SIDXCameraShutterSet**(*SIDXCamera*, *mode*)
status = **SIDXCameraTemperatureExists**(*SIDXCamera*, *available*)
status = **SIDXCameraTemperatureGet**(*SIDXCamera*, *temperature*)
status = **SIDXCameraTransferRateGet**(*SIDXCamera*, *rate*)
status = **SIDXCameraTransferRateSet**(*SIDXCamera*, *transfer_rate*)
status = **SIDXCameraTriggerModeExists**(*SIDXCamera*, *mode*, *available*)
status = **SIDXCameraTriggerModeGet**(*SIDXCamera*, *mode*)
status = **SIDXCameraTriggerModeSet**(*SIDXCamera*, *mode*)
status = **SIDXCameraTriggerSignalExists**(*SIDXCamera*, *mode*, *available*)
status = **SIDXCameraTriggerSignalGet**(*SIDXCamera*, *mode*)
status = **SIDXCameraTriggerSignalSet**(*SIDXCamera*, *mode*)
status = **SIDXDeviceActionDo**(*SIDXDevice*, *setting*, *command*)
status = **SIDXDeviceActionGetByItem**(*SIDXDevice*, *item*, *setting*)
status = **SIDXDeviceActionGetByName**(*SIDXDevice*, *name*, *setting*)
status = **SIDXDeviceActionGetCount**(*SIDXDevice*, *count*)
status = **SIDXDeviceActionGetName**(*SIDXDevice*, *item*, *name*)
status = **SIDXDeviceDriverGetDescription**(*SIDXDevice*, *description*)
status = **SIDXDeviceDriverGetName**(*SIDXDevice*, *name*)
status = **SIDXDeviceDriverGetType**(*SIDXDevice*, *type*)
status = **SIDXDeviceExtraBooleanGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraBooleanSet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraGetByItem**(*SIDXDevice*, *item*, *setting*)
status = **SIDXDeviceExtraGetByName**(*SIDXDevice*, *name*, *setting*)
status = **SIDXDeviceExtraGetCount**(*SIDXDevice*, *count*)
status = **SIDXDeviceExtraGetLabel**(*SIDXDevice*, *setting*, *label*)
status = **SIDXDeviceExtraGetName**(*SIDXDevice*, *item*, *name*)
status = **SIDXDeviceExtraGetType**(*SIDXDevice*, *setting*, *type*)
status = **SIDXDeviceExtraGetUnit**(*SIDXDevice*, *setting*, *unit*)
status = **SIDXDeviceExtraGetValueLocal**(*SIDXDevice*, *setting*, *description*)
status = **SIDXDeviceExtraIntegerGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraIntegerGetRange**(*SIDXDevice*, *setting*, *minimum*, *maximum*)

status = **SIDXDeviceExtraIntegerGetValue**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraIntegerSet**(*SIDXDevice*, *setting*, *integer*)
status = **SIDXDeviceExtraIsSettable**(*SIDXDevice*, *setting*, *settable*)
status = **SIDXDeviceExtraListGet**(*SIDXDevice*, *setting*, *item*)
status = **SIDXDeviceExtraListGetCount**(*SIDXDevice*, *setting*, *count*)
status = **SIDXDeviceExtraListGetEntry**(*SIDXDevice*, *setting*, *item*, *entry*)
status = **SIDXDeviceExtraListGetLocal**(*SIDXDevice*, *setting*, *item*, *description*)
status = **SIDXDeviceExtraListGetValue**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraListSet**(*SIDXDevice*, *setting*, *item*)
status = **SIDXDeviceExtraListSetValue**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraRealGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraRealGetRange**(*SIDXDevice*, *setting*, *minimum*, *maximum*)
status = **SIDXDeviceExtraRealGetValue**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraRealSet**(*SIDXDevice*, *setting*, *real*)
status = **SIDXDeviceExtraSequenceGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraSequenceGetSize**(*SIDXDevice*, *setting*, *size*)
status = **SIDXDeviceExtraSequenceSet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraStringGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraStringSet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceGetDescription**(*SIDXDevice*, *description*)
status = **SIDXDeviceGetLabel**(*SIDXDevice*, *label*)
status = **SIDXDeviceGetName**(*SIDXDevice*, *name*)
status = **SIDXDevicePortAnalogGetRange**(*SIDXDevice*, *port*, *minimum*, *maximum*)
status = **SIDXDevicePortAnalogRead**(*SIDXDevice*, *port*, *voltage*)
status = **SIDXDevicePortAnalogWrite**(*SIDXDevice*, *port*, *voltage*)
status = **SIDXDevicePortBitRead**(*SIDXDevice*, *port*, *asserted*)
status = **SIDXDevicePortBitWrite**(*SIDXDevice*, *port*, *asserted*)
status = **SIDXDevicePortDigitalRead**(*SIDXDevice*, *port*, *value*)
status = **SIDXDevicePortDigitalWrite**(*SIDXDevice*, *port*, *value*)
status = **SIDXDevicePortGetCount**(*SIDXDevice*, *count*)
status = **SIDXDevicePortGetType**(*SIDXDevice*, *port*, *type*)
status = **SIDXGeometryChannelGetDepth**(*SIDXGeometry*, *depth*)
status = **SIDXGeometryImageGetSize**(*SIDXGeometry*, *size*)
status = **SIDXGeometryImageGetType**(*SIDXGeometry*, *type*)
status = **SIDXGeometryPixelGetCount**(*SIDXGeometry*, *x*, *y*)
status = **SIDXGeometryPixelSpacingGet**(*SIDXGeometry*, *x*, *y*)

Methods

SIDXCameraAcquireImageGetLimit

status = **SIDXCameraAcquireImageGetLimit**(*SIDXCamera*, *maximum*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *maximum* A variable to receive an integer value representing the maximum number of images to be acquired.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the maximum number of images that was set by ImageSetLimit.

SIDXCameraAcquireImageSetLimit

status = **SIDXCameraAcquireImageSetLimit**(*SIDXCamera*, *maximum*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *maximum* An integer value specifying the maximum number of images to acquire.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the maximum number of images to be acquired during acquisition. If no value is set, the camera will continuously acquire images until Acquire Stop or Acquire Abort is called.

SIDXCameraAcquireOpen

status = **SIDXCameraAcquireOpen**(*SIDXCamera*, *SIDXAcquire*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *SIDXAcquire* A variable to receive an object to use to aquire images.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Prepare acquisition. The acquisition operates in sequential mode, that is, the acquisition will acquire a specific number of images. This operation does not start Acquire, Acquire must be started explicitly. After this operation, the camera settings cannot be changed, until Acquire ends.

SIDXCameraBinningGet

status = **SIDXCameraBinningGet**(*SIDXCamera*, *x*, *y*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *x* A variable to receive an integer value representing the x axis binning.
- *y* A variable to receive an integer value representing the y axis binning.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the x and y axis binning.

SIDXCameraBinningGetType

status = **SIDXCameraBinningGetType**(*SIDXCamera*, *type*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *type* A variable to receive a value describing the type of binning supported for the camera. The binning type may be none or list. The binning type is never integer or real.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the type of binning for the camera. If the camera binning for the x and y axes interact and must be set as a unit, this is the type of binning for the two axes together. If the camera supports independent selection of binning on the x and y axes, the type of binning will be 'none'. In that case, use the separate methods for the x and y axes to obtain the settable values.

SIDXCameraBinningItemGet

status = **SIDXCameraBinningItemGet**(*SIDXCamera*, *item*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* A variable to receive an integer representing the item of the current binning within the list.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the item of the current binning selection within the binning list.

SIDXCameraBinningItemGetCount

status = **SIDXCameraBinningItemGetCount**(*SIDXCamera*, *count*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *count* A variable to receive an integer count representing the number of (x,y) binning combinations. This value is zero if the camera supports setting the x and y axis binning independently.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the total number of available (x,y) binning combinations. This operation returns a non-zero value only for cameras that support setting the x and y bininng together as a unit. For cameras that support setting the x and y binning independently, the camera reports zero (x,y) binning combinations.

SIDXCameraBinningItemGetEntry

status = **SIDXCameraBinningItemGetEntry**(*SIDXCamera*, *item*, *x*, *y*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* The integer item of the binning selection within the list of available binning choices.
- *x* A variable to receive an integer value representing the x axis binning.
- *y* A variable to receive an integer value representing the y axis binning.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Given the item of a binning selection within the list, returns the x and y axis binning factor.

SIDXCameraBinningItemGetLocal

status = **SIDXCameraBinningItemGetLocal**(*SIDXCamera*, *item*, *description*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* The integer item of the binning selection within the list of available binning choices.
- *description* A variable to receive a text string description of the specified binning list entry.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a text string description of the specified binning setting.

SIDXCameraBinningItemSet

status = **SIDXCameraBinningItemSet**(*SIDXCamera*, *item*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* The integer item of the binning selection within the list of available binning choices.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the binning to the specified item within the binning list.

SIDXCameraBinningSet

status = **SIDXCameraBinningSet**(*SIDXCamera*, *x*, *y*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *x* An integer value that specifies the x axis binning. For example, a value of 2 specifies 2:1 x axis binning, that is, each 2 x axis sensor pixels (columns) are binned and read out as 1 x axis column.
- *y* An integer value that specifes the y axis binning. For example, a value of 4 specifies 4:1 y axis binning, that is, each 4 y axis sensor pixels (rows) are binned and read out as 1 y axis row.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the binning as binning x and y factors. The specific x and y binning combination must be supported by the camera.

SIDXCameraBinningXGetLimit

status = **SIDXCameraBinningXGetLimit**(*SIDXCamera*, *maximum*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *maximum* A variable to receive an integer value representing the maximum x axis binning ratio. This value is 1 or greater.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the largest x axis binning ratio. Use BinningSet to set a value within the limits for the x coordinate.

SIDXCameraBinningXGetType

status = **SIDXCameraBinningXGetType**(*SIDXCamera*, *type*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *type* A variable to receive a value describing the type of binning available. The x axis binning type may be none, list, or integer. The x axis binning type is never a real.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the type of binning selection available for the x axis. The x axis binning type is 'none' if the x and y axis binning interacts and must be set together. If the x axis binning value can be obtained independently, the x axis binning type is not 'none'.

SIDXCameraBinningXItemGetCount

status = **SIDXCameraBinningXItemGetCount**(*SIDXCamera*, *count*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *count* A variable to receive an integer count representing the number of available x axis binning settings.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the count of available x axis binning settings. This method is valid only if the x axis binning type is a list.

SIDXCameraBinningXItemGetEntry

status = **SIDXCameraBinningXItemGetEntry**(*SIDXCamera*, *item*, *binning*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* The integer item of the x axis binning selection within the list of available x axis binning choices.
- *binning* A variable to receive an integer representing the binning ratio. This value is 1 or greater.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the x axis binning ratio for the specified list item. This method is valid only if the x axis binning type is a list. Use BinningSet to set the entry value returned for the x coordinate.

SIDXCameraBinningXItemGetLocal

status = **SIDXCameraBinningXItemGetLocal**(*SIDXCamera*, *item*, *description*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* The integer item of the x axis binning selection within the list of available x axis binning choices.
- *description* A variable to receive a text string description the binning ratio.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a text description of the x axis binning ratio for the specified list item. This method is valid only if the x axis binning type is a list.

SIDXCameraBinningYGetLimit

status = **SIDXCameraBinningYGetLimit**(*SIDXCamera*, *maximum*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *maximum* A variable to receive an integer value representing the maximum y axis binning ratio. This value is 1 or greater.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the maximum y axis binning ratio. The y axis binning ratio can be determined independently of the x axis binning ratio only if the x axis binning type is not 'none'. If the y axis binning ratio can be determined independently, the y axis binning value range is always 1 to the maximum value. Use BinningSet to set a value within the limits for the y coordinate.

SIDXCameraBufferCountGet

status = **SIDXCameraBufferCountGet**(*SIDXCamera*, *count*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *count* A variable to receive an integer value representing the image buffer count setting, as a number of sensor images.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the image buffer count setting. The actual image buffer count may differ from the setting.

SIDXCameraBufferCountSet

status = **SIDXCameraBufferCountSet**(*SIDXCamera*, *count*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *count* An integer value specifying the minimum image buffer size, in units of sensor images.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the minimum number of sensor images to buffer during Acquire. SIDX sets the default sensor image count value to 50.

SIDXCameraClose

status = **SIDXCameraClose**(*SIDXCamera*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Close the camera. After this call, the camera is no longer valid.

SIDXCameraCoolingGet

status = **SIDXCameraCoolingGet**(*SIDXCamera*, *temperature*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *temperature* A variable to receive a real (floating-point) value representing the image sensor target temperature setting, in degrees Celsius.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the image sensor target temperature.

SIDXCameraCoolingGetControl

status = **SIDXCameraCoolingGetControl**(*SIDXCamera*, *control*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *control* A variable to receive a value describing the image sensor temperature control type.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the type of image sensor temperature control and monitoring available.

SIDXCameraCoolingGetRange

status = **SIDXCameraCoolingGetRange**(*SIDXCamera*, *minimum*, *maximum*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *minimum* A variable to receive a real (floating point) value representing the minimum limit.
- *maximum* A variable to receive a real (floating point) value representing the maximum limit.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the minimum and maximum temperature setting. This is not the limits of the sensor temperature, it is the limits of the temperature to which the sensor temperature control can be set.

SIDXCameraCoolingGetValue

status = **SIDXCameraCoolingGetValue**(*SIDXCamera*, *temperature*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *temperature* A variable to receive a real (floating-point) value representing the image sensor temperature.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the image sensor temperature.

SIDXCameraCoolingSet

status = **SIDXCameraCoolingSet**(*SIDXCamera*, *temperature*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *temperature* A real (floating-point) value that specifies the target temperature in degrees Celsius.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the image sensor target temperature. SIDX uses a value valid for the camera that is closest to the setting. Use 'get value' to determine the actual value that is set.

SIDXCameraEMGainGet

status = **SIDXCameraEMGainGet**(*SIDXCamera*, *gain*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *gain* A variable to receive an integer value representing the current EM gain setting.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current EM gain setting. For most cameras with EM gain, the EM gain value is nominal, not an actual gain. The greater the EM gain value, the greater the corresponding gain.

SIDXCameraEMGainGetLabel

status = **SIDXCameraEMGainGetLabel**(*SIDXCamera*, *label*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *label* A variable to receive a text string representing the EM gain label.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the label of the EM gain as a text string.

SIDXCameraEMGainGetRange

status = **SIDXCameraEMGainGetRange**(*SIDXCamera*, *minimum*, *maximum*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *minimum* A variable to receive an integer value representing the minimum limit.
- *maximum* A variable to receive an integer value representing the maximum limit.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the minimum and maximum EM gain value.

SIDXCameraEMGainGetType

status = **SIDXCameraEMGainGetType**(*SIDXCamera*, *type*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *type* A variable to receive a value describing the type of EM gain available. The EM gain setting type is one of none, integer, or list. The EM gain setting type is never real, but this could change in the future.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the type of EM gain setting. The EM gain setting type is one of none, integer, or list.

SIDXCameraEMGainGetUnit

status = **SIDXCameraEMGainGetUnit**(*SIDXCamera*, *unit*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *unit* A variable to receive a text string representing the EM gain unit. If the EM gain is dimensionless, this text string may be blank.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the unit of the EM gain as a text string.

SIDXCameraEMGainGetValue

status = **SIDXCameraEMGainGetValue**(*SIDXCamera*, *gain*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *gain* A variable to receive an integer value representing the current EM gain value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current EM gain value. For most cameras with EM gain, the EM gain value is nominal, not an actual gain. The greater the EM gain value, the greater the corresponding gain.

SIDXCameraEMGainItemGet

status = **SIDXCameraEMGainItemGet**(*SIDXCamera*, *item*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* A variable to receive an integer value representing the current EM gain item within the list.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the item of the EM gain within the gain list.

SIDXCameraEMGainItemGetCount

status = **SIDXCameraEMGainItemGetCount**(*SIDXCamera*, *count*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *count* A variable to receive an integer value representing the count of available EM gain settings.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the count of available EM gain settings. This operation is valid only if the EM gain type is a list.

SIDXCameraEMGainItemGetEntry

status = **SIDXCameraEMGainItemGetEntry**(*SIDXCamera*, *item*, *gain*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* An integer item of the EM gain selection within the list of available EM gain values. The first entry in the list has item zero.
- *gain* A variable to receive an integer value representing the EM gain value corresponding to the item within the list.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the EM gain value corresponding to an item within the list. This operation is valid only if the EM gain type is a list.

SIDXCameraEMGainItemGetLocal

status = **SIDXCameraEMGainItemGetLocal**(*SIDXCamera*, *item*, *description*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* An integer item of the EM gain selection within the list of available EM gain values. The first entry in the list has item zero.
- *description* A variable to receive a text string description of the gain value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a text string description of the EM gain value corresponding to the specified item. This operation is valid only if the EM gain type is a list.

SIDXCameraEMGainItemSet

status = **SIDXCameraEMGainItemSet**(*SIDXCamera*, *item*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* An integer item of the EM gain selection within the list of available EM gain values. The first entry in the list has item zero.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the EM gain to a value corresponding to an item within the list. This operation is valid only if the EM gain type is a list.

SIDXCameraEMGainSet

status = **SIDXCameraEMGainSet**(*SIDXCamera*, *gain*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *gain* An integer value specifying the camera EM gain value to use. This EM gain value must be valid for the camera. The EM gain value is not the actual gain. For example, an EM gain of 4 is a higher gain than an EM gain of 2, but the actual amplification ratio of the two settings may or may not be 4:2.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the EM gain.

SIDXCameraExposeArrayGet

status = **SIDXCameraExposeArrayGet**(*SIDXCamera*, *duration*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *duration* A variable to receive an array of double settings representing the exposure duration settings.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the exposure duration settings as a sequence. This method should be used only if the number of exposure duration values is greater than one (1).

SIDXCameraExposeArrayGetSize

status = **SIDXCameraExposeArrayGetSize**(*SIDXCamera*, *value*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *value* A variable to receive an integer value representing the maximum number of exposure duration intervals supported by the camera.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the maximum number of exposure duration values that can be specified for an exposure sequence. This value is one (1) for most cameras, which support only a single exposure duration. If the value is greater than one, the camera supports multiple exposure duration values in sequence.

SIDXCameraExposeArrayGetValue

status = **SIDXCameraExposeArrayGetValue**(*SIDXCamera*, *duration*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *duration* A variable to receive an array of double values representing the exposure duration values.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the exposure duration values as a sequence. This method should be used only if the number of exposure duration values is greater than one (1).

SIDXCameraExposeArraySet

status = **SIDXCameraExposeArraySet**(*SIDXCamera*, *duration*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *duration* An array containing the exposure duration values to use, in sequence. The exposure duration values are in units of seconds. Expose duration values outside the camera limits will be limited to the exposure time minimum or maximum value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set a sequence of exposure duration values. The length of the sequence must not exceed the maximum number of exposure duration values. This method should be used only if the number of exposure duration values is greater than one (1).

SIDXCameraExposeGet

status = **SIDXCameraExposeGet**(*SIDXCamera*, *duration*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *duration* A variable to receive a real (floating-point) value representing the current exposure duration setting, expressed in seconds.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the exposure time (duration) setting.

SIDXCameraExposeGetRange

status = **SIDXCameraExposeGetRange**(*SIDXCamera*, *minimum*, *maximum*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *minimum* A variable to receive a real (floating point) value representing the minimum limit.
- *maximum* A variable to receive a real (floating point) value representing the maximum limit.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the minimum and maximum exposure time duration. Note that the limits of the exposure time can change based on camera settings.

SIDXCameraExposeGetValue

status = **SIDXCameraExposeGetValue**(*SIDXCamera*, *duration*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *duration* A variable to receive a real (floating-point) value representing the current exposure time duration used by the camera, expressed in seconds.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the exposure time (duration) value used by the camera.

SIDXCameraExposeSet

status = **SIDXCameraExposeSet**(*SIDXCamera*, *duration*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *duration* A real (floating-point) value containing the exposure duration, in seconds. This value must be greater than zero.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the exposure time. If the exposure time is set to a value less than the minimum, or greater than the maximum, the exposure time used will be limited to the exposure time minimum or maximum value.

SIDXCameraExternalDelayGet

status = **SIDXCameraExternalDelayGet**(*SIDXCamera*, *interval*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *interval* A variable to receive a real (floating-point) value specifying external time interval delay in seconds.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the time interval in seconds required by the external device between the end of an exposure and the start of the next exposure. This is the minimum delay, the actual delay may be longer, depending on the camera.

SIDXCameraExternalDelaySet

status = **SIDXCameraExternalDelaySet**(*SIDXCamera*, *interval*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *interval* A real (floating-point) value specifying the external time interval delay in seconds.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the time interval in seconds between the end of an exposure and the start of the next exposure required by the external device. If the external delay is not set, the default is zero.

SIDXCameraFanControlExists

status = **SIDXCameraFanControlExists**(*SIDXCamera*, *available*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *available* A variable to receive a boolean value, true if computer-based fan control is available, false if not.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Determine if fan control is available.

SIDXCameraFanControlGet

status = **SIDXCameraFanControlGet**(*SIDXCamera*, *enable*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *enable* A variable to receive a boolean value, true to enable the fan, false to disable the fan.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain whether or not fan control is enabled. This operation will report an error if the camera does not have computer-based fan control.

SIDXCameraFanControlSet

status = **SIDXCameraFanControlSet**(*SIDXCamera*, *enable*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *enable* A boolean value, true to enable the fan, false to disable the fan.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the fan control. This operation will report an error if the camera does not have computer-based fan control. If the fan control is enabled, the fan may turn on for cooling. If the fan control is disabled, the fan is prevented from turning on.

SIDXCameraGainGet

status = **SIDXCameraGainGet**(*SIDXCamera*, *gain*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *gain* A variable to receive a real (floating-point) value representing the analog gain setting. This setting is always greater than zero. If the camera does not have an analog gain setting, this value is 1.0.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current readout analog gain setting.

SIDXCameraGainGetLabel

status = **SIDXCameraGainGetLabel**(*SIDXCamera*, *label*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *label* A variable to receive a text string representing the label for the analog gain.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the camera analog gain label as a text string. The label is independent of the gain value, for example, it might be the string "Gain".

SIDXCameraGainGetRange

status = **SIDXCameraGainGetRange**(*SIDXCamera*, *minimum*, *maximum*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *minimum* A variable to receive a real (floating point) value representing the minimum limit.
- *maximum* A variable to receive a real (floating point) value representing the maximum limit.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the analog gain minimum and maximum value.

SIDXCameraGainGetType

status = **SIDXCameraGainGetType**(*SIDXCamera*, *type*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *type* A variable to receive a value describing the type of analog gain available. The type is one of none, list, or real. The type is never integer.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the type of analog gain setting.

SIDXCameraGainGetUnit

status = **SIDXCameraGainGetUnit**(*SIDXCamera*, *unit*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *unit* A variable to receive a text string representing the unit for the analog gain. If the gain is dimensionless, this string may be empty.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the camera analog gain unit as a text string. The unit is independent of the gain value.

SIDXCameraGainGetValue

status = **SIDXCameraGainGetValue**(*SIDXCamera*, *gain*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *gain* A variable to receive a real (floating-point) value representing the analog gain value. This value is always greater than zero. If the camera does not have an analog gain setting, this value is 1.0.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current readout analog gain value used by the camera.

SIDXCameraGainItemGet

status = **SIDXCameraGainItemGet**(*SIDXCamera*, *item*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* A variable to receive an integer value representing the item of the gain within the list of analog gains.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the item of the gain within the gain list.

SIDXCameraGainItemGetCount

status = **SIDXCameraGainItemGetCount**(*SIDXCamera*, *count*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *count* A variable to receive an integer representing the count of available gain settings.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the count of available gain settings. This operation is valid only if the gain type is list.

SIDXCameraGainItemGetEntry

status = **SIDXCameraGainItemGetEntry**(*SIDXCamera*, *item*, *gain*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* An integer item of the gain selection within the list of available gain values. The first entry in the list has item zero.
- *gain* A variable to receive a real (floating-point) value representing the gain. This value is always greater than zero.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the gain value corresponding to a specified gain list item. This operation is valid only if the gain type is list.

SIDXCameraGainItemGetLocal

status = **SIDXCameraGainItemGetLocal**(*SIDXCamera*, *item*, *description*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* An integer item of the gain selection within the list of available gain values. The first entry in the list has item zero.
- *description* A variable to receive a text string description of the gain.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a text description of the specified gain list item. This operation is valid only if the gain type is list.

SIDXCameraGainItemSet

status = **SIDXCameraGainItemSet**(*SIDXCamera*, *item*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* An integer item of the gain selection within the list of available gain values. The first entry in the list has item zero.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the analog gain according to the specified gain list item. This operation is valid only if the gain type is list.

SIDXCameraGainSet

status = **SIDXCameraGainSet**(*SIDXCamera*, *gain*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *gain* A real (floating-point) value specifying the camera analog gain value to use. This analog gain value must be valid for the camera.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the camera analog gain to the specified value.

SIDXCameraGetLastError

SIDXCameraGetLastError(*SIDXCamera*, *message*)

Parameters

- *SIDXCamera* A handle value that references the SIDX camera context.
- *message* A text string to receive the text associated with the error code specified.

The function translates the error code into a text string.

SIDXCameraIntensifierGet

status = **SIDXCameraIntensifierGet**(*SIDXCamera*, *gain*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *gain* A variable to receive a real (floating-point) value representing the intensifier gain setting.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current intensifier setting.

SIDXCameraIntensifierGetLabel

status = **SIDXCameraIntensifierGetLabel**(*SIDXCamera*, *label*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *label* A variable to receive a text string representing the intensifier setting label.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the label for the intensifier setting.

SIDXCameraIntensifierGetRange

status = **SIDXCameraIntensifierGetRange**(*SIDXCamera*, *minimum*, *maximum*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *minimum* A variable to receive a real (floating point) value representing the minimum limit.
- *maximum* A variable to receive a real (floating point) value representing the maximum limit.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the minimum and maximum intensifier value.

SIDXCameraIntensifierGetType

status = **SIDXCameraIntensifierGetType**(*SIDXCamera*, *type*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *type* A variable to receive a value describing the type of intensifier gain setting. The intensifier gain type is none or real. The intensifier gain type is never list or integer.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the type of intensifier setting.

SIDXCameraIntensifierGetUnit

status = **SIDXCameraIntensifierGetUnit**(*SIDXCamera*, *unit*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *unit* A variable to receive a text string representing the intensifier gain unit. The intensifier gain may be dimensionless, in which case this text string may be blank.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the intensifier gain unit as a text string.

SIDXCameraIntensifierGetValue

status = **SIDXCameraIntensifierGetValue**(*SIDXCamera*, *gain*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *gain* A variable to receive a real (floating-point) value representing the intensifier gain value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current intensifier value.

SIDXCameraIntensifierSet

status = **SIDXCameraIntensifierSet**(*SIDXCamera*, *gain*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *gain* A real (floating-point) value specifying the intensifier gain. The value must be a valid intensifier gain setting.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the intensifier.

SIDXCameraOperateGet

status = **SIDXCameraOperateGet**(*SIDXCamera*, *value*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *value* A variable to receive a text string value representing the operating mode setting. The value should be consistent, so it should be possible restore the camera to the same setting later by setting the camera to this value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current operating mode setting as a camera-specific text string.

SIDXCameraOperateItemGet

status = **SIDXCameraOperateItemGet**(*SIDXCamera*, *item*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* A variable to receive an integer item representing the operating mode setting.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current operating mode setting.

SIDXCameraOperateItemGetCount

status = **SIDXCameraOperateItemGetCount**(*SIDXCamera*, *count*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *count* A variable to receive an integer value representing the count of available operating mode settings.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the count of available camera operating modes. The result is one (1) if the camera offers only one Operate mode, that is, no choice of Operate mode.

SIDXCameraOperateItemGetLocal

status = **SIDXCameraOperateItemGetLocal**(*SIDXCamera*, *item*, *description*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* An integer item of the operating mode selection within the list of available operating mode values. The first entry has item zero.
- *description* A variable to receive a text string description of the operating mode.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a text string description of the operating mode corresponding to the specified item.

SIDXCameraOperateItemSet

status = **SIDXCameraOperateItemSet**(*SIDXCamera*, *item*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* An integer item of the operating mode. The first operating mode has the item value zero (0);

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the operating mode to the specified item.

SIDXCameraOperateSet

status = **SIDXCameraOperateSet**(*SIDXCamera*, *value*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *value* The text string operating mode value to set. The value operating mode must be valid for the camera. The value is camera-specific.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the current operating mode setting, using a camera-specific text string. When the caller sets the operation mode, all other parameters are reset to their default values. For example, the readout mode will be reset to the value that represents the slowest readout rate for the highest pixel depth.

SIDXCameraPollingGet

status = **SIDXCameraPollingGet**(*SIDXCamera*, *interval*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *interval* A variable to receive a real (floating-point) value representing the service interval setting, in seconds.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the service interval (polling interval) setting.

SIDXCameraPollingSet

status = **SIDXCameraPollingSet**(*SIDXCamera*, *interval*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *interval* A real (floating-point) value specifying the service interval, in seconds.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the service interval during Acquire. The service interval is the maximum time between calls to the Acquire 'GetStatus' operation. The longer the service interval, the larger the buffer needed to hold images during Acquire. In some cases, the actual maximum service interval may differ significantly from the setting. SIDX sets the default polling value to 1 second.

SIDXCameraReadoutGet

status = **SIDXCameraReadoutGet**(*SIDXCamera*, *value*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *value* A variable to receive a text string value representing the setting. The value is camera-specific, the same value may represent a different setting on different cameras. However, the value should be consistent, so it should be possible restore the camera to the same setting later by setting the camera to this value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current Readout setting as a camera-specific value.

SIDXCameraReadoutGetValue

status = **SIDXCameraReadoutGetValue**(*SIDXCamera*, *pixel_rate*, *pixel_depth*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *pixel_rate* A variable to receive a real (floating-point) value representing the nominal pixel readout rate of images acquired using the readout, measured in Hz. For example, if the pixel readout rate is 10MHz, this value is 10 million (1.0e+7)
- *pixel_depth* A variable to receive an integer value representing the pixel depth of images acquired using the readout, measured in bits.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the pixel depth and pixel rate of images acquired using the Readout, with the current settings.

SIDXCameraReadoutItemGet

status = **SIDXCameraReadoutItemGet**(*SIDXCamera*, *item*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* A variable to receive an integer item representing the Readout setting.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current Readout setting.

SIDXCameraReadoutItemGetCount

status = **SIDXCameraReadoutItemGetCount**(*SIDXCamera*, *count*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *count* A variable to receive an integer value representing the count of available Readout settings.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the count of available Readout settings. The result is one (1) if the camera offers only one Readout, that is, no choice of Readout. If the camera offers a choice of Readout, then ReadoutItem zero (0) generally produces the highest-quality images, and the highest-numbered Readout generally produces images at the highest speed.

SIDXCameraReadoutItemGetEntry

status = **SIDXCameraReadoutItemGetEntry**(*SIDXCamera*, *item*, *pixel_rate*, *pixel_depth*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* An integer item of the Readout selection within the list of available Readout values. The first entry has item zero.
- *pixel_rate* A variable to receive a real (floating-point) value representing the nominal pixel readout rate of images acquired using the readout, measured in Hz. For example, if the pixel readout rate is 10MHz, this value is 10 million (1.0e+7)
- *pixel_depth* A variable to receive an integer value representing the pixel depth of images acquired using the readout, measured in bits.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the pixel depth and pixel rate of images acquired using the Readout corresponding to the specified item.

SIDXCameraReadoutItemGetLocal

status = **SIDXCameraReadoutItemGetLocal**(*SIDXCamera*, *item*, *description*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* An integer item of the Readout selection within the list of available Readout values. The first entry has item zero.
- *description* A variable to receive a text string description of the Readout.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a text string description of the Readout corresponding to the specified item.

SIDXCameraReadoutItemSet

status = **SIDXCameraReadoutItemSet**(*SIDXCamera*, *item*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *item* An integer item of the Readout. The first Readout has the item value zero (0);

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the Readout to the specified item.

SIDXCameraReadoutSet

status = **SIDXCameraReadoutSet**(*SIDXCamera*, *value*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *value* The text string value to set. The Readout setting value must be valid for the camera. The value is camera-specific.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the current Readout setting, using a camera-specific value. The available Readout modes are dependent on the Operate mode that is set. The default Readout mode is the one that provides the slowest readout rate for the highest pixel depth.

SIDXCameraROIClear

status = **SIDXCameraROIClear**(*SIDXCamera*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Clear the ROI (region of interest), so the entire image is acquired.

SIDXCameraROIGet

status = **SIDXCameraROIGet**(*SIDXCamera*, *x1*, *y1*, *x2*, *y2*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *x1* A variable to receive an integer value representing the x1 value, that is, the left side of the region.
- *y1* A variable to receive an integer value representing the y1 value, that is, the top of the region.
- *x2* A variable to receive an integer value representing the x2 value, that is, the right side of the region.
- *y2* A variable to receive an integer value representing the y2 value, that is, the bottom of the region.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the ROI setting.

SIDXCameraROIGetValue

status = **SIDXCameraROIGetValue**(*SIDXCamera*, *x1*, *y1*, *x2*, *y2*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *x1* A variable to receive an integer value representing the x1 value, that is, the left side of the region.
- *y1* A variable to receive an integer value representing the y1 value, that is, the top of the region.
- *x2* A variable to receive an integer value representing the x2 value, that is, the right side of the region.
- *y2* A variable to receive an integer value representing the y2 value, that is, the bottom of the region.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the ROI used. The ROI used may be larger than the ROI setting, if the camera imposes restrictions on the ROI. For example, if the camera requires that the ROI x position and x width be multiples of 4, an ROI that begins at x position 5 will be adjusted to begin at 4.

SIDXCameraROISet

status = **SIDXCameraROISet**(*SIDXCamera*, *x1*, *y1*, *x2*, *y2*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *x1* An integer value specifying the left-side x coordinate. The x coordinate is measured from left to right.
- *y1* An integer value specifying the y coordinate. The y coordinate is measured from top to bottom.
- *x2* An integer value specifying the right-side x coordinate. The x coordinate is measured from left to right. The region width in pixels is x2-x1, so the region excludes the x2 column.
- *y2* An integer value specifying the bottom y coordinate. The y coordinate is measured from top to bottom. The region height in pixels is y2-y1, so the region excludes the y2 row.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the ROI (region of interest) for imaging. The ROI is specified in sensor coordinates, not binned coordinates. For example, if a 1000x1000 sensor is binned 4x4 to yield a 250x250 image, the ROI is interpreted in the 1000x1000 sensor coordinates, not the 250x250 binned coordinates.

SIDXCameraRotateClear

status = **SIDXCameraRotateClear**(*SIDXCamera*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Clear the rotation and mirroring settings, so acquired images are not transformed.

SIDXCameraRotateMirrorX

status = **SIDXCameraRotateMirrorX**(*SIDXCamera*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Mirror the image in x. Calls to this method are cumulative, so two successive calls result in no mirroring. The mirroring and rotation calls operate in sequence. Calling RotateSet followed RotateMirrorX rotates the image, then mirrors the image. Calling RotateMirrorX then RotateSet mirrors the image, then rotates the image.

SIDXCameraRotateMirrorY

status = **SIDXCameraRotateMirrorY**(*SIDXCamera*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Mirror the image in y. Calls to this method are cumulative, so two successive calls result in no mirroring. The mirroring and rotation calls operate in sequence. Calling RotateSet followed RotateMirrorY rotates the image, then mirrors the image. Calling RotateMirrorY then RotateSet mirrors the image, then rotates the image.

SIDXCameraRotateSet

status = **SIDXCameraRotateSet**(*SIDXCamera*, *count*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *count* An integer value specifying the image rotation as a signed count of 90 degree clockwise rotations.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the image rotation as a count of 90 degree clockwise rotations. For example, a 90 degree clockwise rotation is 1, a 90 degree counterclockwise rotation is -1, and a 360 degree rotation is 4. Rotations are cumulative, so successive calls add. For example, four calls each with a rotation count of 1 produce a rotation count of 4.

SIDXCameraShutterExists

status = **SIDXCameraShutterExists**(*SIDXCamera*, *mode*, *available*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *mode* A value describing the shutter control mode value to test.
- *available* A variable to receive a boolean value, true if the shutter control mode is available, false if the shutter control mode is not available.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Determine whether a given shutter control mode is available.

SIDXCameraShutterGet

status = **SIDXCameraShutterGet**(*SIDXCamera*, *mode*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *mode* A variable to receive a value describing the current shutter control setting mode.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current shutter control setting.

SIDXCameraShutterSet

status = **SIDXCameraShutterSet**(*SIDXCamera*, *mode*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *mode* A value describing the shutter control mode to use.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the shutter control to the specified value.

SIDXCameraTemperatureExists

status = **SIDXCameraTemperatureExists**(*SIDXCamera*, *available*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *available* A variable to receive a boolean value, true if the sensor temperature is available, false if the sensor temperature is not available.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Determine whether the camera sensor temperature is available.

SIDXCameraTemperatureGet

status = **SIDXCameraTemperatureGet**(*SIDXCamera*, *temperature*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *temperature* A variable to receive a real (floating-point) value representing the temperature of the sensor.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current measured sensor temperature.

SIDXCameraTransferRateGet

status = **SIDXCameraTransferRateGet**(*SIDXCamera*, *rate*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *rate* A variable to receive a real (floating-point) value specifying the tranfer rate in bytes per second.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the transfer rate in bytes per second.

SIDXCameraTransferRateSet

status = **SIDXCameraTransferRateSet**(*SIDXCamera*, *transfer_rate*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *transfer_rate* A real (floating-point) value specifying the transfer rate in bytes per second.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the transfer rate in bytes per second. This parameter is used to calculate the time required to transfer an image from the camera to the computer. If the value is not set, the default transfer rate is zero (0).

SIDXCameraTriggerModeExists

status = **SIDXCameraTriggerModeExists**(*SIDXCamera*, *mode*, *available*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *mode* The trigger input control mode to test.
- *available* A variable to receive a boolean value, true if the trigger input control mode is available, false if the trigger input control mode is not available.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Determine whether a given trigger input control mode is available.

SIDXCameraTriggerModeGet

status = **SIDXCameraTriggerModeGet**(*SIDXCamera*, *mode*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *mode* A variable to receive a value describing the current trigger input control setting mode.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current trigger input control setting.

SIDXCameraTriggerModeSet

status = **SIDXCameraTriggerModeSet**(*SIDXCamera*, *mode*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *mode* A value describing the trigger input control value to use.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the trigger input control to the specified value.

SIDXCameraTriggerSignalExists

status = **SIDXCameraTriggerSignalExists**(*SIDXCamera*, *mode*, *available*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *mode* The trigger input signal mode to test.
- *available* A variable to receive a boolean value, true if the trigger input signal mode is available, false if the trigger input signal mode is not available.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Determine whether a given trigger input signal mode is available.

SIDXCameraTriggerSignalGet

status = **SIDXCameraTriggerSignalGet**(*SIDXCamera*, *mode*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *mode* A variable to receive a value describing the current trigger input signal setting mode.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current trigger input signal setting.

SIDXCameraTriggerSignalSet

status = **SIDXCameraTriggerSignalSet**(*SIDXCamera*, *mode*)

Parameters

- *SIDXCamera* A handle value that references the SIDXCamera context
- *mode* A value describing the trigger input signal value to use.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the trigger input signal to the specified value.

4.3 SIDXAcquire

An object of this interface represents image acquisition in SIDX.

Reference

status = **SIDXAcquireAbort**(*SIDXAcquire*)
status = **SIDXAcquireArchiveIsWriting**(*SIDXAcquire*, *writing*)
status = **SIDXAcquireArchiveOpenNew**(*SIDXAcquire*, *path*, *data_type*, *overwrite*, *SIDXArchive*)
status = **SIDXAcquireArchiveWrite**(*SIDXAcquire*, *image_index*, *image_count*)
status = **SIDXAcquireClose**(*SIDXAcquire*)
status = **SIDXAcquireDisplayOpen**(*SIDXAcquire*, *output_image_type*, *SIDXDisplay*)
status = **SIDXAcquireGetBufferCount**(*SIDXAcquire*, *count*)
status = **SIDXAcquireGetGapInterval**(*SIDXAcquire*, *interval*)
status = **SIDXAcquireGetImageInterval**(*SIDXAcquire*, *interval*)
status = **SIDXAcquireGetPollingInterval**(*SIDXAcquire*, *interval*)
status = **SIDXAcquireGetReadoutInterval**(*SIDXAcquire*, *interval*)
status = **SIDXAcquireGetStatus**(*SIDXAcquire*, *acquiring*)
status = **SIDXAcquireImageGetCount**(*SIDXAcquire*, *count*)
status = **SIDXAcquireImageGetDescription**
(*SIDXAcquire*, *image_index*, *start_time*, *exposure_duration*)
status = **SIDXAcquireRead**(*SIDXAcquire*, *image_count*, *image_data*)
status = **SIDXAcquireReadGetPosition**(*SIDXAcquire*, *position*)
status = **SIDXAcquireReadoutExists**(*SIDXAcquire*, *available*)
status = **SIDXAcquireReadSetPosition**(*SIDXAcquire*, *image_index*)
status = **SIDXAcquireSpacingGetSize**(*SIDXAcquire*, *size*)
status = **SIDXAcquireStart**(*SIDXAcquire*)
status = **SIDXAcquireStop**(*SIDXAcquire*)
status = **SIDXGeometryChannelGetDepth**(*SIDXGeometry*, *depth*)
status = **SIDXGeometryImageGetSize**(*SIDXGeometry*, *size*)
status = **SIDXGeometryImageGetType**(*SIDXGeometry*, *type*)
status = **SIDXGeometryPixelGetCount**(*SIDXGeometry*, *x*, *y*)
status = **SIDXGeometryPixelSpacingGet**(*SIDXGeometry*, *x*, *y*)

Methods

SIDXAcquireAbort

status = **SIDXAcquireAbort**(*SIDXAcquire*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Interrupt and terminate image Acquire. The caller can perform this operation an arbitrary number of times.

SIDXAcquireArchiveIsWriting

status = **SIDXAcquireArchiveIsWriting**(*SIDXAcquire*, *writing*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *writing* A variable to receive an boolean value, true if writing images, false if not.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Determine if issued writing to file operations are being completed. If an error occurs while archiving, all further write operations will report an error.

SIDXAcquireArchiveOpenNew

status = **SIDXAcquireArchiveOpenNew**(*SIDXAcquire*, *path*, *data_type*, *overwrite*, *SIDXArchive*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *path* A text string specifying the path to the archive file to create.
- *data_type* A text string specifying the type of archive data file to write. This value must be the text string "TIFF".
- *overwrite* A boolean value, true if SIDX should overwrite the existing file, false if it should not.
- *SIDXArchive* A variable to receive an object to use to access the image archive.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Create an file as an image archive. If the file already exists and the 'overwrite' parameter is set to 'true', the existing file will be destroyed. If the file already exists and the 'overwrite' parameter is set to 'false', SIDX returns an error.

SIDXAcquireArchiveWrite

status = **SIDXAcquireArchiveWrite**(*SIDXAcquire*, *image_index*, *image_count*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *image_index* An integer value specifying the starting image number to archive.
- *image_count* An integer value specifying the number of images to archive.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Archive the specified images. This operation is valid only if an archive is open. The operation is synchronous, it will not return until the images have been archived or an error occurs. All images to be archived must exist.

SIDXAcquireClose

status = **SIDXAcquireClose**(*SIDXAcquire*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Close the Acquire. After this call, the Acquire is no longer valid and camera settings can be changed.

SIDXAcquireDisplayOpen

status = **SIDXAcquireDisplayOpen**(*SIDXAcquire*, *output_image_type*, *SIDXDisplay*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *output_image_type* A value describing the type of output image to display.
- *SIDXDisplay* A variable to receive an object to use to access the presentation display.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Prepare the transformation for display data. Display can be used to prepare the image data from the camera for display.

SIDXAcquireGetBufferCount

status = **SIDXAcquireGetBufferCount**(*SIDXAcquire*, *count*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *count* A variable to receive an integer value representing the image buffer count, as a number of images.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the image buffer count used, that is, the number of images that can be stored in the image buffer. This value may differ from the setting.

SIDXAcquireGetGapInterval

status = **SIDXAcquireGetGapInterval**(*SIDXAcquire*, *interval*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *interval* A variable to receive a real (floating-point) value specifying the minimum time interval gap between the end of one exposure and the start of the next.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the minimum time interval from the end of an exposure to the start of the next exposure.

SIDXAcquireGetImageInterval

status = **SIDXAcquireGetImageInterval**(*SIDXAcquire*, *interval*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *interval* A variable to receive a real (floating-point) value specifying the image interval in seconds.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the time interval between successive images. This value is an estimate based on the camera settings. For cameras in modes with controlled timing, this value may be accurate. For example, if the camera supports fixed-rate image streaming with overlapped image integration and readout, this value may be the actual interval between successive images. For external trigger modes, the reported interval is the minimum time from the start of one image exposure to the start of the next. The actual interval is determined by the external trigger. The image interval is constant during Acquire, it does not change.

SIDXAcquireGetPollingInterval

status = **SIDXAcquireGetPollingInterval**(*SIDXAcquire*, *interval*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *interval* A variable to receive a real (floating-point) value representing the maximum service interval, in seconds.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the maximum interval between successive calls to the Acquire 'GetStatus' call during image Acquire. For example, if the maximum service interval is 250ms, during Acquire, the Acquire 'GetStatus' operation must be called at least every 250ms.

SIDXAcquireGetReadoutInterval

status = **SIDXAcquireGetReadoutInterval**(*SIDXAcquire*, *interval*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *interval* A variable to receive a real (floating-point) value specifying the readout interval in seconds.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the readout interval of an exposed image in seconds.

SIDXAcquireGetStatus

status = **SIDXAcquireGetStatus**(*SIDXAcquire*, *acquiring*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *acquiring* A variable to receive a boolean value, true if the camera is acquiring, false if not.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Update the Acquire status. This is a polling operation. The operation must be called at least once within the service interval. For example, if the Acquire service interval is 500ms, this operation must be called at least every 500ms. The function returns true or false to indicate that the camera is actually acquiring. If an error occurs, than the boolean value is undefined.

SIDXAcquireImageGetCount

status = **SIDXAcquireImageGetCount**(*SIDXAcquire*, *count*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *count* A variable to receive an integer count containing the total number of images acquired since Acquire started.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the total number of acquired images since last call to Acquire Start.

SIDXAcquireImageGetDescription

status = **SIDXAcquireImageGetDescription**
(*SIDXAcquire*, *image_index*, *start_time*, *exposure_duration*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *image_index* An integer value specifying the image for which to return time.
- *start_time* A variable to receive a real (floating-point) value specifying the start time.
- *exposure_duration* A variable to receive a real (floating-point) value specifying the exposure duration.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtains the exposure duration and start time for a specified image. If the camera does not provide a start time, SIDX will provide the software time of the received image. The exposure duration is constant throughout an acquisition, except in the case that the exposure duration is set by an external trigger signal. In that case, the camera may provide the actual exposure duration separately for each image and SIDX will return a value of zero (0). If the camera does not provide the exposure duration, SIDX will return the exposure duration that was set. If the camera does provide the exposure duration, SIDX will return the value from the camera.

SIDXAcquireRead

status = **SIDXAcquireRead**(*SIDXAcquire*, *image_count*, *image_data*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *image_count* An integer value specifying the number of images to read.
- *image_data* The array to receive the image data.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Read images starting from the current read position. The number of images read is the same as the image_count, otherwise SIDX will report an error.

SIDXAcquireReadGetPosition

status = **SIDXAcquireReadGetPosition**(*SIDXAcquire*, *position*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *position* A variable to receive an integer value representing the read position.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current read position. The read position is an integer image index. The first image is at position zero.

SIDXAcquireReadoutExists

status = **SIDXAcquireReadoutExists**(*SIDXAcquire*, *available*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *available* A variable to receive a boolean value, true if the readout interval is available, false if the readout interval is not available.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Determine whether readout interval information is available.

SIDXAcquireReadSetPosition

status = **SIDXAcquireReadSetPosition**(*SIDXAcquire*, *image_index*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *image_index* An integer value specifying new read position, expressed as an image number. The image number is greater than or equal to zero.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the read position to the specific image.

SIDXAcquireSpacingGetSize

status = **SIDXAcquireSpacingGetSize**(*SIDXAcquire*, *size*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context
- *size* A variable to receive an integer value representing the size of an image, measured in bytes.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the size of an image. The size of an image is the amount of storage required for an image during acquisition, including padding. For example, if images during acquisition are padded to the next multiple of 512 bytes, this value will always be a multiple of 512

SIDXAcquireStart

status = **SIDXAcquireStart**(*SIDXAcquire*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Start Acquire. The caller can perform this operation an arbitrary number of times.

SIDXAcquireStop

status = **SIDXAcquireStop**(*SIDXAcquire*)

Parameters

- *SIDXAcquire* A handle value that references the SIDXAcquire context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Stop Acquire. This operation should be used only if Acquire has completed as expected, the operation should not be used to interrupt running image Acquire. The caller can perform this operation an arbitrary number of times.

4.4 SIDXArchive

An object of this interface represents an open image archive in SIDX.

Reference

status = **SIDXArchiveClose**(*SIDXArchive*)
status = **SIDXArchiveDisplayOpen**(*SIDXArchive*, *output_image_type*, *SIDXDisplay*)
status = **SIDXArchiveImageGetCount**(*SIDXArchive*, *count*)
status = **SIDXArchiveRead**(*SIDXArchive*, *image_count*, *image_data*)
status = **SIDXArchiveReadGetPosition**(*SIDXArchive*, *position*)
status = **SIDXArchiveReadSetPosition**(*SIDXArchive*, *image_index*)
status = **SIDXArchiveSpacingGetSize**(*SIDXArchive*, *size*)
status = **SIDXGeometryChannelGetDepth**(*SIDXGeometry*, *depth*)
status = **SIDXGeometryImageGetSize**(*SIDXGeometry*, *size*)
status = **SIDXGeometryImageGetType**(*SIDXGeometry*, *type*)
status = **SIDXGeometryPixelGetCount**(*SIDXGeometry*, *x*, *y*)
status = **SIDXGeometryPixelSpacingGet**(*SIDXGeometry*, *x*, *y*)

Methods

SIDXArchiveClose

status = **SIDXArchiveClose**(*SIDXArchive*)

Parameters

- *SIDXArchive* A handle value that references the SIDXArchive context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Close the archive. After this call, the archive is no longer valid.

SIDXArchiveDisplayOpen

status = **SIDXArchiveDisplayOpen**(*SIDXArchive*, *output_image_type*, *SIDXDisplay*)

Parameters

- *SIDXArchive* A handle value that references the SIDXArchive context
- *SIDXDisplay* A variable to receive an object to use to access the presentation display.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Prepare the transformation for display data. Display can be used to prepare the image data from the archive file for display.

SIDXArchiveImageGetCount

status = **SIDXArchiveImageGetCount**(*SIDXArchive*, *count*)

Parameters

- *SIDXArchive* A handle value that references the SIDXArchive context
- *count* A variable to receive an integer value representing the count of images in the archive. If the archive is empty, this value is zero.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the count of images in the archive.

SIDXArchiveRead

status = **SIDXArchiveRead**(*SIDXArchive*, *image_count*, *image_data*)

Parameters

- *SIDXArchive* A handle value that references the SIDXArchive context
- *image_count* An integer number of images to read. This value must be greater than zero. The read must not extend past the end of the archive.
- *image_data* An array to receive the image data.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Read image data from the archive, starting at the current read position. The images read include any padding.

SIDXArchiveReadGetPosition

status = **SIDXArchiveReadGetPosition**(*SIDXArchive*, *position*)

Parameters

- *SIDXArchive* A handle value that references the SIDXArchive context
- *position* A variable to receive an integer value representing the current read position. The read position for the first image is zero.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current read position.

SIDXArchiveReadSetPosition

status = **SIDXArchiveReadSetPosition**(*SIDXArchive*, *image_index*)

Parameters

- *SIDXArchive* A handle value that references the SIDXArchive context
- *image_index* An integer value that specifies the new read position, expressed in units of images. The read position must be within the archive, that is, the position must be zero or greater, and cannot exceed the count of images in the archive.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the read position to the specified value.

SIDXArchiveSpacingGetSize

status = **SIDXArchiveSpacingGetSize**(*SIDXArchive*, *size*)

Parameters

- *SIDXArchive* A handle value that references the SIDXArchive context
- *size* A variable to receive an integer value representing the size of an image, measured in bytes.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the size of an image. The size of an image is the amount of storage required for an image in the archive, including padding. For example, if images in an archive are padded to the next multiple of 512 bytes, this value will always be a multiple of 512

4.5 SIDXStage

An object of this interface represents an open stage in SIDX.

Reference

status = **SIDXStageAccelerateGetLimitXY**(*SIDXStage*, *maximum*)
status = **SIDXStageAccelerateGetLimitZ**(*SIDXStage*, *maximum*)
status = **SIDXStageAccelerateGetXY**(*SIDXStage*, *acceleration*)
status = **SIDXStageAccelerateGetZ**(*SIDXStage*, *acceleration*)
status = **SIDXStageAccelerateSetXY**(*SIDXStage*, *acceleration*)
status = **SIDXStageAccelerateSetZ**(*SIDXStage*, *acceleration*)
status = **SIDXStageAxisEnableXY**(*SIDXStage*, *x_enable*, *y_enable*)
status = **SIDXStageAxisEnableZ**(*SIDXStage*, *z_enable*)
status = **SIDXStageAxisExistsXY**(*SIDXStage*, *motorized*)
status = **SIDXStageAxisExistsZ**(*SIDXStage*, *motorized*)
status = **SIDXStageBacklashExists**(*SIDXStage*, *control*)
status = **SIDXStageBacklashGetXY**(*SIDXStage*, *distance*)
status = **SIDXStageBacklashGetZ**(*SIDXStage*, *distance*)
status = **SIDXStageBacklashSetXY**(*SIDXStage*, *distance*)
status = **SIDXStageBacklashSetZ**(*SIDXStage*, *distance*)
status = **SIDXStageClose**(*SIDXStage*)
status = **SIDXStageJoystickDisable**(*SIDXStage*)
status = **SIDXStageJoystickEnable**(*SIDXStage*)
status = **SIDXStageJoystickSetMovementXY**(*SIDXStage*, *x_right*, *y_forward*, *fraction*)
status = **SIDXStageJoystickSetMovementZ**(*SIDXStage*, *z_up*, *fraction*)
status = **SIDXStageLimitIsXMinus**(*SIDXStage*, *limit*)
status = **SIDXStageLimitIsXPlus**(*SIDXStage*, *limit*)
status = **SIDXStageLimitIsXY**(*SIDXStage*, *limit*)
status = **SIDXStageLimitIsXYZ**(*SIDXStage*, *limit*)
status = **SIDXStageLimitIsYMinus**(*SIDXStage*, *limit*)
status = **SIDXStageLimitIsYPlus**(*SIDXStage*, *limit*)
status = **SIDXStageLimitIsZ**(*SIDXStage*, *limit*)
status = **SIDXStageLimitIsZMinus**(*SIDXStage*, *limit*)
status = **SIDXStageLimitIsZPlus**(*SIDXStage*, *limit*)
status = **SIDXStageLimitQuery**(*SIDXStage*)
status = **SIDXStageLoaderBarcodeExists**(*SIDXStage*, *installed*)
status = **SIDXStageLoaderBarcodeGet**(*SIDXStage*, *barcode*)
status = **SIDXStageLoaderExists**(*SIDXStage*, *installed*)
status = **SIDXStageLoaderHeapExists**(*SIDXStage*, *heap*, *installed*)
status = **SIDXStageLoaderHeapGetCount**(*SIDXStage*, *count*)
status = **SIDXStageLoaderHeapSlotGetCount**(*SIDXStage*, *slots*)
status = **SIDXStageLoaderInitialize**(*SIDXStage*)

status = **SIDXStageLoaderLoad**(*SIDXStage*, *heap*, *slot*)
status = **SIDXStageLoaderMoveStage**(*SIDXStage*)
status = **SIDXStageLoaderMovingIs**(*SIDXStage*, *moving*)
status = **SIDXStageLoaderScan**(*SIDXStage*, *heap*)
status = **SIDXStageLoaderScanExists**(*SIDXStage*, *available*)
status = **SIDXStageLoaderScanGetResults**(*SIDXStage*, *heap*, *status*)
status = **SIDXStageLoaderScanningIs**(*SIDXStage*, *scanning*)
status = **SIDXStageLoaderStop**(*SIDXStage*)
status = **SIDXStageLoaderUnload**(*SIDXStage*, *heap*, *slot*)
status = **SIDXStageMotorAccelerateGet**(*SIDXStage*, *motor*, *acceleration*)
status = **SIDXStageMotorAccelerateSet**(*SIDXStage*, *motor*, *acceleration*)
status = **SIDXStageMotorBacklashExists**(*SIDXStage*, *motor*, *available*)
status = **SIDXStageMotorBacklashGet**(*SIDXStage*, *motor*, *distance*)
status = **SIDXStageMotorBacklashSet**(*SIDXStage*, *motor*, *distance*)
status = **SIDXStageMotorExists**(*SIDXStage*, *motor*, *installed*)
status = **SIDXStageMotorGetByName**(*SIDXStage*, *name*, *motor*)
status = **SIDXStageMotorGetCount**(*SIDXStage*, *identifiers*)
status = **SIDXStageMotorGetName**(*SIDXStage*, *motor*, *name*)
status = **SIDXStageMotorLimit**(*SIDXStage*, *motor*, *limit*)
status = **SIDXStageMotorLimitMinus**(*SIDXStage*, *motor*, *limit*)
status = **SIDXStageMotorLimitPlus**(*SIDXStage*, *motor*, *limit*)
status = **SIDXStageMotorMoveRelative**(*SIDXStage*, *motor*, *offset*)
status = **SIDXStageMotorMoveSpeed**(*SIDXStage*, *motor*, *velocity*)
status = **SIDXStageMotorMoveStop**(*SIDXStage*, *motor*)
status = **SIDXStageMotorMovingIs**(*SIDXStage*, *motor*, *moving*)
status = **SIDXStageMotorSpeedGet**(*SIDXStage*, *motor*, *speed*)
status = **SIDXStageMotorSpeedSet**(*SIDXStage*, *motor*, *speed*)
status = **SIDXStageMovePositionXY**(*SIDXStage*, *x_position_m*, *y_position_m*)
status = **SIDXStageMovePositionXYZ**(*SIDXStage*, *x_position_m*, *y_position_m*, *z_position_m*)
status = **SIDXStageMovePositionZ**(*SIDXStage*, *z_position_m*)
status = **SIDXStageMoveRelativeXY**(*SIDXStage*, *x_offset_m*, *y_offset_m*)
status = **SIDXStageMoveRelativeXYZ**(*SIDXStage*, *x_offset_m*, *y_offset_m*, *z_offset_m*)
status = **SIDXStageMoveRelativeZ**(*SIDXStage*, *z_offset_m*)
status = **SIDXStageMoveSpeedXY**(*SIDXStage*, *x_velocity_mps*, *y_velocity_mps*)
status = **SIDXStageMoveSpeedXYZ**(*SIDXStage*, *x_velocity_mps*, *y_velocity_mps*, *z_velocity_mps*)
status = **SIDXStageMoveSpeedZ**(*SIDXStage*, *z_velocity_mps*)
status = **SIDXStageMoveStopXYZ**(*SIDXStage*)
status = **SIDXStageMovingIsXY**(*SIDXStage*, *moving*)
status = **SIDXStageMovingIsXYZ**(*SIDXStage*, *moving*)
status = **SIDXStageMovingIsZ**(*SIDXStage*, *moving*)
status = **SIDXStageMovingQuery**(*SIDXStage*)
status = **SIDXStageOriginResetXYZ**(*SIDXStage*)
status = **SIDXStagePositionGet**(*SIDXStage*, *position_x*, *position_y*, *position_z*)
status = **SIDXStagePositionGetX**(*SIDXStage*, *coordinate*)
status = **SIDXStagePositionGetY**(*SIDXStage*, *coordinate*)

status = **SIDXStagePositionGetZ**(*SIDXStage*, *coordinate*)
status = **SIDXStagePositionQuery**(*SIDXStage*)
status = **SIDXStageReset**(*SIDXStage*)
status = **SIDXStageResolutionGetXY**(*SIDXStage*, *resolution*)
status = **SIDXStageResolutionGetZ**(*SIDXStage*, *resolution*)
status = **SIDXStageRotateClear**(*SIDXStage*)
status = **SIDXStageRotateMirrorX**(*SIDXStage*)
status = **SIDXStageRotateMirrorY**(*SIDXStage*)
status = **SIDXStageRotateSet**(*SIDXStage*, *count*)
status = **SIDXStageSpeedGetXY**(*SIDXStage*, *speed*)
status = **SIDXStageSpeedGetZ**(*SIDXStage*, *speed*)
status = **SIDXStageSpeedSetXY**(*SIDXStage*, *speed*)
status = **SIDXStageSpeedSetZ**(*SIDXStage*, *speed*)
status = **SIDXDeviceActionDo**(*SIDXDevice*, *setting*, *command*)
status = **SIDXDeviceActionGetByItem**(*SIDXDevice*, *item*, *setting*)
status = **SIDXDeviceActionGetByName**(*SIDXDevice*, *name*, *setting*)
status = **SIDXDeviceActionGetCount**(*SIDXDevice*, *count*)
status = **SIDXDeviceActionGetName**(*SIDXDevice*, *item*, *name*)
status = **SIDXDeviceDriverGetDescription**(*SIDXDevice*, *description*)
status = **SIDXDeviceDriverGetName**(*SIDXDevice*, *name*)
status = **SIDXDeviceDriverGetType**(*SIDXDevice*, *type*)
status = **SIDXDeviceExtraBooleanGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraBooleanSet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraGetByItem**(*SIDXDevice*, *item*, *setting*)
status = **SIDXDeviceExtraGetByName**(*SIDXDevice*, *name*, *setting*)
status = **SIDXDeviceExtraGetCount**(*SIDXDevice*, *count*)
status = **SIDXDeviceExtraGetLabel**(*SIDXDevice*, *setting*, *label*)
status = **SIDXDeviceExtraGetName**(*SIDXDevice*, *item*, *name*)
status = **SIDXDeviceExtraGetType**(*SIDXDevice*, *setting*, *type*)
status = **SIDXDeviceExtraGetUnit**(*SIDXDevice*, *setting*, *unit*)
status = **SIDXDeviceExtraGetValueLocal**(*SIDXDevice*, *setting*, *description*)
status = **SIDXDeviceExtraIntegerGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraIntegerGetRange**(*SIDXDevice*, *setting*, *minimum*, *maximum*)
status = **SIDXDeviceExtraIntegerGetValue**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraIntegerSet**(*SIDXDevice*, *setting*, *integer*)
status = **SIDXDeviceExtraIsSettable**(*SIDXDevice*, *setting*, *settable*)
status = **SIDXDeviceExtraListGet**(*SIDXDevice*, *setting*, *item*)
status = **SIDXDeviceExtraListGetCount**(*SIDXDevice*, *setting*, *count*)
status = **SIDXDeviceExtraListGetEntry**(*SIDXDevice*, *setting*, *item*, *entry*)
status = **SIDXDeviceExtraListGetLocal**(*SIDXDevice*, *setting*, *item*, *description*)
status = **SIDXDeviceExtraListGetValue**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraListSet**(*SIDXDevice*, *setting*, *item*)
status = **SIDXDeviceExtraListSetValue**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraRealGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraRealGetRange**(*SIDXDevice*, *setting*, *minimum*, *maximum*)

status = **SIDXDeviceExtraRealGetValue**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraRealSet**(*SIDXDevice*, *setting*, *real*)
status = **SIDXDeviceExtraSequenceGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraSequenceGetSize**(*SIDXDevice*, *setting*, *size*)
status = **SIDXDeviceExtraSequenceSet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraStringGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraStringSet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceGetDescription**(*SIDXDevice*, *description*)
status = **SIDXDeviceGetLabel**(*SIDXDevice*, *label*)
status = **SIDXDeviceGetName**(*SIDXDevice*, *name*)
status = **SIDXDevicePortAnalogGetRange**(*SIDXDevice*, *port*, *minimum*, *maximum*)
status = **SIDXDevicePortAnalogRead**(*SIDXDevice*, *port*, *voltage*)
status = **SIDXDevicePortAnalogWrite**(*SIDXDevice*, *port*, *voltage*)
status = **SIDXDevicePortBitRead**(*SIDXDevice*, *port*, *asserted*)
status = **SIDXDevicePortBitWrite**(*SIDXDevice*, *port*, *asserted*)
status = **SIDXDevicePortDigitalRead**(*SIDXDevice*, *port*, *value*)
status = **SIDXDevicePortDigitalWrite**(*SIDXDevice*, *port*, *value*)
status = **SIDXDevicePortGetCount**(*SIDXDevice*, *count*)
status = **SIDXDevicePortGetType**(*SIDXDevice*, *port*, *type*)

Methods

SIDXStageAccelerateGetLimitXY

status = **SIDXStageAccelerateGetLimitXY**(*SIDXStage*, *maximum*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *maximum* A variable to receive a real (floating-point) value representing the acceleration. If -1 is returned, the maximum acceleration is unknown.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the maximum acceleration during stage movement for the X and Y axes in meters/second^2.

SIDXStageAccelerateGetLimitZ

status = **SIDXStageAccelerateGetLimitZ**(*SIDXStage*, *maximum*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *maximum* A variable to receive a real (floating-point) value representing the acceleration. If -1 is returned, the maximum acceleration is unknown.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the maximum acceleration during stage movement for the Z axis in meters/second^2.

SIDXStageAccelerateGetXY

status = **SIDXStageAccelerateGetXY**(*SIDXStage*, *acceleration*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *acceleration* A variable to receive a real (floating-point) value representing the acceleration fraction, a value in the range [0, 1.0].

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the acceleration during stage movement for the X and Y axes as a fraction of maximum acceleration.

SIDXStageAccelerateGetZ

status = **SIDXStageAccelerateGetZ**(*SIDXStage*, *acceleration*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *acceleration* A variable to receive a real (floating-point) value representing the acceleration fraction, a value in the range [0, 1.0].

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the acceleration during stage movement for the Z axis as a fraction of maximum acceleration.

SIDXStageAccelerateSetXY

status = **SIDXStageAccelerateSetXY**(*SIDXStage*, *acceleration*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *acceleration* The desired acceleration fraction, a value in the range [0, 1.0].

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the acceleration during stage movement for the X and Y axes as a fraction of maximum acceleration.

SIDXStageAccelerateSetZ

status = **SIDXStageAccelerateSetZ**(*SIDXStage*, *acceleration*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *acceleration* The desired acceleration fraction,a value in the range [0, 1.0].

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the acceleration during stage movement for the Z axis as a fraction of maximum acceleration.

SIDXStageAxisEnableXY

status = **SIDXStageAxisEnableXY**(*SIDXStage*, *x_enable*, *y_enable*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *x_enable* True to enable the axis motor, false to disable.
- *y_enable* True to enable the axis motor, false to disable.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Enables and disables motorized control of the stage axes.

SIDXStageAxisEnableZ

status = **SIDXStageAxisEnableZ**(*SIDXStage*, *z_enable*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *z_enable* True to enable the axis motor, false to disable.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Enables and disables motorized control of the focus axis.

SIDXStageAxisExistsXY

status = **SIDXStageAxisExistsXY**(*SIDXStage*, *motorized*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motorized* A variable to receive a boolean value, true if X and Y are motorized axes, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the motorized status of the X and Y axes.

SIDXStageAxisExistsZ

status = **SIDXStageAxisExistsZ**(*SIDXStage*, *motorized*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motorized* A variable to receive a boolean value, true if Z is motorized, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the motorized status of the Z axis.

SIDXStageBacklashExists

status = **SIDXStageBacklashExists**(*SIDXStage*, *control*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *control* A variable to receive a boolean value specifying the availability of backlash control.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether backlash control is available on this stage.

SIDXStageBacklashGetXY

status = **SIDXStageBacklashGetXY**(*SIDXStage*, *distance*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *distance* A variable to receive a real (floating-point) value representing the backlash distance in meters.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the backlash distance for the X and Y axes, measured in meters.

SIDXStageBacklashGetZ

status = **SIDXStageBacklashGetZ**(*SIDXStage*, *distance*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *distance* A variable to receive a real (floating-point) value representing the backlash distance.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the backlash distance for the Z axis, measured in meters.

SIDXStageBacklashSetXY

status = **SIDXStageBacklashSetXY**(*SIDXStage*, *distance*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *distance* The backlash distance.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the backlash distance for the X and Y axes, measured in meters.

SIDXStageBacklashSetZ

status = **SIDXStageBacklashSetZ**(*SIDXStage*, *distance*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *distance* The backlash distance.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the backlash distance for the Z axis, measured in meters.

SIDXStageClose

status = **SIDXStageClose**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Close the stage controller.

SIDXStageJoystickDisable

status = **SIDXStageJoystickDisable**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the stage's attached joystick to be disabled.

SIDXStageJoystickEnable

status = **SIDXStageJoystickEnable**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the stage's attached joystick to be enabled.

SIDXStageJoystickSetMovementXY

status = **SIDXStageJoystickSetMovementXY**(*SIDXStage*, *x_right*, *y_forward*, *fraction*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *x_right* True if, as the x axis coordinate increases, the stage moves the right. False if it moves left.
- *y_right* True if, as the y axis coordinate increases, the stage moves the forward. False if it moves backward.
- *fraction* The stage motion speed to use, as a as a fraction of full speed. This is a numeric value in the range [0,1.0].

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the direction and speed of the X and Y axis controls for stage's attached joystick.

SIDXStageJoystickSetMovementZ

status = **SIDXStageJoystickSetMovementZ**(*SIDXStage*, *z_up*, *fraction*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *direction* True if, as the z axis coordinate increases, the stage moves the up. False if it moves down.
- *fraction* The stage motion speed to use, as a as a fraction of full speed. This is a numeric value in the range [0,1.0].

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the direction and speed of the Z axis control for stage's attached joystick.

SIDXStageLimitIsXMinus

status = **SIDXStageLimitIsXMinus**(*SIDXStage*, *limit*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *limit* A variable to receive a boolean value, true if the lower X axis limit switch was set, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether the lower limit switch associated with the X axis was set during the last successful limit query.

SIDXStageLimitIsXPlus

status = **SIDXStageLimitIsXPlus**(*SIDXStage*, *limit*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *limit* A variable to receive a boolean value, true if the upper X axis limit switch was set, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether the upper limit switch associated with the X axis was set during the last successful limit query.

SIDXStageLimitIsXY

status = **SIDXStageLimitIsXY**(*SIDXStage*, *limit*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *limit* A variable to receive a boolean value, true if one of the X or Y axes' limit switches was set, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether any of the limit switches associated with the X or Y axes were set during the last successful limit query.

SIDXStageLimitIsXYZ

status = **SIDXStageLimitIsXYZ**(*SIDXStage*, *limit*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *limit* A variable to receive a boolean value, true if one of the X, Y or Z axes' limit switches was set, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether any of the limit switches associated with the X, Y or Z axes were set during the last successful limit query.

SIDXStageLimitIsYMinus

status = **SIDXStageLimitIsYMinus**(*SIDXStage*, *limit*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *limit* A variable to receive a boolean value, true if the lower Y axis limit switch was set, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether the lower limit switch associated with the Y axis was set during the last successful limit query.

SIDXStageLimitIsYPlus

status = **SIDXStageLimitIsYPlus**(*SIDXStage*, *limit*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *limit* A variable to receive a boolean value, true if the upper Y axis limit switch was set, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether the upper limit switch associated with the Y axis was set during the last successful limit query.

SIDXStageLimitIsZ

status = **SIDXStageLimitIsZ**(*SIDXStage*, *limit*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *limit* A variable to receive a boolean value, true if one of the Z axis limit switches was set, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether either of the limit switches associated with the Z axis were set during the last successful limit query.

SIDXStageLimitIsZMinus

status = **SIDXStageLimitIsZMinus**(*SIDXStage*, *limit*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *limit* A variable to receive a boolean value, true if the lower Z axis limit switch was set, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether the lower limit switch associated with the Z axis was set during the last successful limit query.

SIDXStageLimitIsZPlus

status = **SIDXStageLimitIsZPlus**(*SIDXStage*, *limit*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *limit* A variable to receive a boolean value, true if the upper Z axis limit switch was set, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether the upper limit switch associated with the Z axis was set during the last successful limit query.

SIDXStageLimitQuery

status = **SIDXStageLimitQuery**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Queries the current stage limit switch values. Each axis has two limit switches, the values of all limit switches are captured synchronously.

SIDXStageLoaderBarcodeExists

status = **SIDXStageLoaderBarcodeExists**(*SIDXStage*, *installed*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *installed* A variable to receive a boolean value, true if a barcode reader is installed, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Returns whether or not a barcode reader is available on the loader.

SIDXStageLoaderBarcodeGet

status = **SIDXStageLoaderBarcodeGet**(*SIDXStage*, *barcode*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *barcode* A variable to receive a text string representing the barcode on the slide or plate.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Returns the barcode of the slide or plate currently on the stage.

SIDXStageLoaderExists

status = **SIDXStageLoaderExists**(*SIDXStage*, *installed*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *installed* A variable to receive a boolean value, true is a loader is installed, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Returns whether a slide/plate loader is attached and available for control.

SIDXStageLoaderHeapExists

status = **SIDXStageLoaderHeapExists**(*SIDXStage*, *heap*, *installed*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *heap* An integer value of the heap number.
- *installed* A variable to receive a boolean value, true if the specified heap is installed.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Queries the loader to determine whether a particular heap is installed.

SIDXStageLoaderHeapGetCount

status = **SIDXStageLoaderHeapGetCount**(*SIDXStage*, *count*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *count* A variable to receive an integer count of supported heaps.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Returns the number of heaps that are potentially supported by this loader.

SIDXStageLoaderHeapSlotGetCount

status = **SIDXStageLoaderHeapSlotGetCount**(*SIDXStage*, *slots*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *slots* A variable to receive an integer value representing the number of slots in a heap.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

LoaderHeapSlotCount returns the number of slots in a heap. Any results returned by Loader-ScanGetResults will have this number of elements.

SIDXStageLoaderInitialize

status = **SIDXStageLoaderInitialize**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the loader to move to its home position and initialize any necessary parameters. This command must be called before beginning operations and after every LoaderStop.

SIDXStageLoaderLoad

status = **SIDXStageLoaderLoad**(*SIDXStage*, *heap*, *slot*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *heap* An integer value representing the heap where the slide/plate is to be found.
- *slot* An integer value representing the slot position where the slide/plate is to be found.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the loader to move the specified slide/plate from the specified slot to the stage.

SIDXStageLoaderMoveStage

status = **SIDXStageLoaderMoveStage**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to position itself for the loader to place or remove a slide/plate. This command must be called immediately before LoaderLoad or LoaderUnload. If the stage is not positioned correctly those functions will return with an error. Failing to call this function and to wait for it to finish execution before loading or returning a slide/plate may result in damage to the stage, slide/plate or loader.

SIDXStageLoaderMovingIs

status = **SIDXStageLoaderMovingIs**(*SIDXStage*, *moving*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *moving* A variable to receive a boolean value, true if the loader is currently moving, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Returns whether the loader is currently moving.

SIDXStageLoaderScan

status = **SIDXStageLoaderScan**(*SIDXStage*, *heap*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *heap* An integer value representing the heap to be scanned.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the Loader to begin scanning the specified heap for filled slots.

SIDXStageLoaderScanExists

status = **SIDXStageLoaderScanExists**(*SIDXStage*, *available*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *available* A variable to receive a boolean value, true if the loader scanning is available, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Returns whether the loader can scan a heap to search for filled slots.

SIDXStageLoaderScanGetResults

status = **SIDXStageLoaderScanGetResults**(*SIDXStage*, *heap*, *status*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *heap* An integer value representing the specified heap.
- *status* A variable to receive an array of boolean value representing the filled status of all the slots.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Returns a boolean array showing the filled status of all the slots on the specified cassette heap.

SIDXStageLoaderScanningIs

status = **SIDXStageLoaderScanningIs**(*SIDXStage*, *scanning*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *scanning* A variable to receive a boolean value, true if the loader is scanning, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Returns whether the loader is currently scanning.

SIDXStageLoaderStop

status = **SIDXStageLoaderStop**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the loader to halt all activities. Before resuming normal operations after a stop, the user must call LoaderInitialize.

SIDXStageLoaderUnload

status = **SIDXStageLoaderUnload**(*SIDXStage*, *heap*, *slot*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *heap* An integer value representing the heap where the slide/plate is to be taken to.
- *slot* An integer value representing the slot position where the slide/plate is to be taken to.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the loader to unload a slide/plate from the stage and return it to the specified heap and position.

SIDXStageMotorAccelerateGet

status = **SIDXStageMotorAccelerateGet**(*SIDXStage*, *motor*, *acceleration*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.
- *acceleration* A variable to receive a real (floating-point) value representing the acceleration fraction, a value in the range [0, 1.0].

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the acceleration during stage movement for the specified motor as a fraction of maximum acceleration.

SIDXStageMotorAccelerateSet

status = **SIDXStageMotorAccelerateSet**(*SIDXStage*, *motor*, *acceleration*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.
- *acceleration* An integer value representing the desired acceleration fraction, a value in the range [0, 1.0].

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the acceleration during stage movement for the specified motor as a fraction of maximum acceleration.

SIDXStageMotorBacklashExists

status = **SIDXStageMotorBacklashExists**(*SIDXStage*, *motor*, *available*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.
- *available* A variable to receive a boolean value, true if backlash control is available, otherwise false.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether backlash control is available on the specified motor.

SIDXStageMotorBacklashGet

status = **SIDXStageMotorBacklashGet**(*SIDXStage*, *motor*, *distance*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.
- *distance* A variable to receive a real (floating-point) value representing the backlash distance.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the backlash distance for the specified motor, in meters.

SIDXStageMotorBacklashSet

status = **SIDXStageMotorBacklashSet**(*SIDXStage*, *motor*, *distance*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.
- *distance* A real (floating-point) value representing the backlash distance.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the backlash distance for the specified motor, in meters.

SIDXStageMotorExists

status = **SIDXStageMotorExists**(*SIDXStage*, *motor*, *installed*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.
- *installed* A variable to receive a boolean value, true if the motor is installed, otherwise false.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether the specified motor is currently installed.

SIDXStageMotorGetByName

status = **SIDXStageMotorGetByName**(*SIDXStage*, *name*, *motor*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *name* An integer value representing the name of the motor.
- *motor* A variable to receive an integer handle to the motor.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the handle to the specified motor.

SIDXStageMotorGetCount

status = **SIDXStageMotorGetCount**(*SIDXStage*, *identifiers*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *identifiers* A variable to receive an integer value representing the number of identifiers.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the number of different motor identifiers.

SIDXStageMotorGetName

status = **SIDXStageMotorGetName**(*SIDXStage*, *motor*, *name*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *name* A variable to receive a n integer value representing the name of the motor.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the name corresponding to the specified motor number.

SIDXStageMotorLimit

status = **SIDXStageMotorLimit**(*SIDXStage*, *motor*, *limit*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.
- *limit* A variable to receive a boolean value, true if one of the stepper motor's limit switches is set, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether either of the limit switches associated with the specified motor is currently set.

SIDXStageMotorLimitMinus

status = **SIDXStageMotorLimitMinus**(*SIDXStage*, *motor*, *limit*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.
- *limit* A variable to receive a boolean value, true if the lower motor limit switch is set, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether the lower limit switch associated with the specified motor is currently set.

SIDXStageMotorLimitPlus

status = **SIDXStageMotorLimitPlus**(*SIDXStage*, *motor*, *limit*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.
- *limit* A variable to receive a boolean value, true if the upper motor limit switch is set, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets whether the upper limit switch associated with the specified motor is currently set.

SIDXStageMotorMoveRelative

status = **SIDXStageMotorMoveRelative**(*SIDXStage*, *motor*, *offset*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.
- *offset* A real (floating-point) value representing the offset to move (in the units expected by the stage). This can be a positive or negative value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to begin moving the specified motor the specified offset, measured in meters.

SIDXStageMotorMoveSpeed

status = **SIDXStageMotorMoveSpeed**(*SIDXStage*, *motor*, *velocity*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.
- *velocityA* real (floating-point) value representing the speed. This can be a positive or negative value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to begin moving the specified motor at the specified speed, as a fraction of maximum speed.

SIDXStageMotorMoveStop

status = **SIDXStageMotorMoveStop**(*SIDXStage*, *motor*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to stop the motor.

SIDXStageMotorMovingIs

status = **SIDXStageMotorMovingIs**(*SIDXStage*, *motor*, *moving*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.
- *moving* A variable to receive a boolean value, true if the motor is moving, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the motion status of the specified motor.

SIDXStageMotorSpeedGet

status = **SIDXStageMotorSpeedGet**(*SIDXStage*, *motor*, *speed*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.
- *speed* A variable to receive a real (floating-point) value representing the speed fraction, a value in the range [0, 1.0].

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the speed for the specified motor as a fraction of maximum speed.

SIDXStageMotorSpeedSet

status = **SIDXStageMotorSpeedSet**(*SIDXStage*, *motor*, *speed*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *motor* An integer value representing the handle to the motor.
- *speed* An real (floating-point) value representing the speed fraction, a value in the range [0, 1.0].

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the speed for the specified motor as a fraction of maximum speed.

SIDXStageMovePositionXY

status = **SIDXStageMovePositionXY**(*SIDXStage*, *x_position_m*, *y_position_m*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *x_position_m* A real (floating-point) value representing the coordinate on the X axis to move to, in meters.
- *y_position_m* A real (floating-point) value representing the coordinate on the Y axis to move to, in meters.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to begin moving to the absolute position specified along the X and Y axes, measured in meters.

SIDXStageMovePositionXYZ

status = **SIDXStageMovePositionXYZ**(*SIDXStage*, *x_position_m*, *y_position_m*, *z_position_m*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *x_position_m* A real (floating-point) value representing the coordinate on the X axis to move to, in meters.
- *y_position_m* A real (floating-point) value representing the coordinate on the Y axis to move to, in meters.
- *z_position_m* A real (floating-point) value representing the coordinate on the Z axis to move to, in meters.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to begin moving to the absolute position specified along the X, Y and Z axes, measured in meters.

SIDXStageMovePositionZ

status = **SIDXStageMovePositionZ**(*SIDXStage*, *z_position_m*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *z_position_m* A real (floating-point) value representing the coordinate on the Z axis to move to, in meters.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to begin moving to the absolute position specified along the Z axis, measured in meters.

SIDXStageMoveRelativeXY

status = **SIDXStageMoveRelativeXY**(*SIDXStage*, *x_offset_m*, *y_offset_m*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *x_offset_m* A real (floating-point) value representing the X axis offset to move in meters. This can be a positive or negative value.
- *y_offset_m* A real (floating-point) value representing the Y axis offset to move in meters. This can be a positive or negative value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to begin moving the specified offsets along the X and Y axes, measured in meters.

SIDXStageMoveRelativeXYZ

status = **SIDXStageMoveRelativeXYZ**(*SIDXStage*, *x_offset_m*, *y_offset_m*, *z_offset_m*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *x_offset_m* A real (floating-point) value representing the X axis offset to move in meters. This can be a positive or negative value.
- *y_offset_m* A real (floating-point) value representing the Y axis offset to move in meters. This can be a positive or negative value.
- *z_offset_m* A real (floating-point) value representing the Z axis offset to move in meters. This can be a positive or negative value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to begin moving the specified offsets along the X, Y and Z axes, measured in meters.

SIDXStageMoveRelativeZ

status = **SIDXStageMoveRelativeZ**(*SIDXStage*, *z_offset_m*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *z_offset_m* A real (floating-point) value representing the offset to move in meters. This can be a positive or negative value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to begin moving the specified offset along the Z axis, measured in meters.

SIDXStageMoveSpeedXY

status = **SIDXStageMoveSpeedXY**(*SIDXStage*, *x_velocity_mps*, *y_velocity_mps*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *x_velocity_mps* A real (floating-point) value representing the X axis speed. This can be a positive or negative value.
- *y_velocity_mps* A real (floating-point) value representing the Y axis speed. This can be a positive or negative value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to begin moving at the specified speed along the X and Y axes, measured in meters/second.

SIDXStageMoveSpeedXYZ

status = **SIDXStageMoveSpeedXYZ**(*SIDXStage*, *x_velocity_mps*, *y_velocity_mps*, *z_velocity_mps*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *x_velocity_mps* A real (floating-point) value representing the X axis speed. This can be a positive or negative value.
- *y_velocity_mps* A real (floating-point) value representing the Y axis speed. This can be a positive or negative value.
- *z_velocity_mps* A real (floating-point) value representing the Z axis speed. This can be a positive or negative value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to begin moving at the specified speed along the X, Y and Z axes, measured in meters/second.

SIDXStageMoveSpeedZ

status = **SIDXStageMoveSpeedZ**(*SIDXStage*, *z_velocity_mps*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *z_velocity_mps* A real (floating-point) value representing the Z axis speed. This can be a positive or negative value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to begin moving at the specified speed along the Z axis, measured in meters/second.

SIDXStageMoveStopXYZ

status = **SIDXStageMoveStopXYZ**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to stop all movement along the X, Y and Z axes.

SIDXStageMovingIsXY

status = **SIDXStageMovingIsXY**(*SIDXStage*, *moving*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *moving* A variable to receive a boolean value, true if the stage is moving along the X or Y axes, false if X and Y are not moving.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the motion status of the X and Y axes during the last successful motion query.

SIDXStageMovingIsXYZ

status = **SIDXStageMovingIsXYZ**(*SIDXStage*, *moving*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *moving* A variable to receive a boolean value, true if the stage is moving along the X, Y or Z axes, false if X, Y and Z are not moving.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the motion status of the X, Y and Z axes during the last successful motion query.

SIDXStageMovingIsZ

status = **SIDXStageMovingIsZ**(*SIDXStage*, *moving*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *moving* A variable to receive a boolean value, true if the stage is moving along the Z axis, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the motion status of the Z axis during the last successful motion query.

SIDXStageMovingQuery

status = **SIDXStageMovingQuery**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Queries the stage about the current motion along the X, Y and Z axes. The results of query are accessed through any of the MovingIs... methods.

SIDXStageOriginResetXYZ

status = **SIDXStageOriginResetXYZ**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the current position to zero on the X, Y and Z axes. Future absolute positions are referenced to this new origin.

SIDXStagePositionGet

status = **SIDXStagePositionGet**(*SIDXStage*, *position_x*, *position_y*, *position_z*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *coordinates* A variable to receive an object of double settings containing the X, Y and Z coordinates.
- *position_x* A variable to receive a real (floating-point) value representing the x position.
- *position_y* A variable to receive a real (floating-point) value representing the y position.
- *position_z* A variable to receive a real (floating-point) value representing the z position.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the current absolute position coordinates for all three axes, in meters.

SIDXStagePositionGetX

status = **SIDXStagePositionGetX**(*SIDXStage*, *coordinate*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *coordinate* A variable to receive a real (floating-point) value representing the X axis coordinate in meters.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the current absolute position coordinate for the X axis, in meters.

SIDXStagePositionGetY

status = **SIDXStagePositionGetY**(*SIDXStage*, *coordinate*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *coordinate* A variable to receive a real (floating-point) value representing the Y axis coordinate in meters.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the current absolute position coordinate for the Y axis, in meters.

SIDXStagePositionGetZ

status = **SIDXStagePositionGetZ**(*SIDXStage*, *coordinate*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *coordinate* A variable to receive a real (floating-point) value representing the Z axis coordinate in meters.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the current absolute position coordinate for the Z axis, in meters.

SIDXStagePositionQuery

status = **SIDXStagePositionQuery**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Queries the stage about the current position of the X, Y and Z axes. The results of query are accessed through any of the PositionGet... methods.

SIDXStageReset

status = **SIDXStageReset**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Commands the stage to stop all motion and reboot.

SIDXStageResolutionGetXY

status = **SIDXStageResolutionGetXY**(*SIDXStage*, *resolution*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *resolution* A variable to receive a real (floating-point) value representing the resolution.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the resolution of the X and Y axes in units of meters.

SIDXStageResolutionGetZ

status = **SIDXStageResolutionGetZ**(*SIDXStage*, *resolution*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *resolution* A variable to receive a real (floating-point) value representing the resolution.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the resolution of the Z axis in units of meters.

SIDXStageRotateClear

status = **SIDXStageRotateClear**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Clear the rotation and mirroring settings, so acquired images are not transformed.

SIDXStageRotateMirrorX

status = **SIDXStageRotateMirrorX**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Mirror the image in x. Calls to this method are cumulative, so two successive calls result in no mirroring. The mirroring and rotation calls operate in sequence. Calling RotateSet followed RotateMirrorX rotates the image, then mirrors the image. Calling RotateMirrorX then RotateSet mirrors the image, then rotates the image.

SIDXStageRotateMirrorY

status = **SIDXStageRotateMirrorY**(*SIDXStage*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Mirror the image in y. Calls to this method are cumulative, so two successive calls result in no mirroring. The mirroring and rotation calls operate in sequence. Calling RotateSet followed RotateMirrorY rotates the image, then mirrors the image. Calling RotateMirrorY then RotateSet mirrors the image, then rotates the image.

SIDXStageRotateSet

status = **SIDXStageRotateSet**(*SIDXStage*, *count*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *count* An integer value specifying the image rotation as a signed count of 90 degree clockwise rotations.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the image rotation as a count of 90 degree clockwise rotations. For example, a 90 degree clockwise rotation is 1, a 90 degree counterclockwise rotation is -1, and a 360 degree rotation is 4. Rotations are cumulative, so successive calls add. For example, four calls each with a rotation count of 1 produce a rotation count of 4.

SIDXStageSpeedGetXY

status = **SIDXStageSpeedGetXY**(*SIDXStage*, *speed*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *speed* A variable to receive a real (floating-point) value representing the speed in meters per second.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the speed for the X and Y axes in meters per second.

SIDXStageSpeedGetZ

status = **SIDXStageSpeedGetZ**(*SIDXStage*, *speed*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *speed* A variable to receive a real (floating-point) value representing the speed in meters per second.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the speed for the Z axis in meters per second.

SIDXStageSpeedSetXY

status = **SIDXStageSpeedSetXY**(*SIDXStage*, *speed*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *speed* An integer value representing the speed in meters per second.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the speed for the X and Y axes in meters per second.

SIDXStageSpeedSetZ

status = **SIDXStageSpeedSetZ**(*SIDXStage*, *speed*)

Parameters

- *SIDXStage* A handle value that references the SIDXStage context
- *speed* An integer value representing the speed in meters per second.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the speed for the Z axis in meters per second.

4.6 SIDXDisplay

An object of this class represents image display in SIDX.

Reference

status = **SIDXDisplayClose**(*SIDXDisplay*)
SIDXDisplayGetLastError(*SIDXDisplay*, *message*)
status = **SIDXDisplayGetDisplay**(*SIDXDisplay*, *image_data*, *display*)
status = **SIDXGeometryChannelGetDepth**(*SIDXGeometry*, *depth*)
status = **SIDXGeometryImageGetSize**(*SIDXGeometry*, *size*)
status = **SIDXGeometryImageGetType**(*SIDXGeometry*, *type*)
status = **SIDXGeometryPixelGetCount**(*SIDXGeometry*, *x*, *y*)
status = **SIDXGeometryPixelSpacingGet**(*SIDXGeometry*, *x*, *y*)

Methods

SIDXDisplayClose

status = **SIDXDisplayClose**(*SIDXDisplay*)

Parameters
- *SIDXDisplay* A handle value that references the SIDXDisplay context

Return value
- *status* Non-zero if the function failed, zero if it succeeded.

Close the display. After this call, the display is no longer valid.

SIDXDisplayGetLastError

SIDXDisplayGetLastError(*SIDXDisplay*, *message*)

Parameters
- *SIDXDisplay* A handle value that references the SIDX display context.
- *message* A text string to receive the text associated with the error code specified.

The function translates the error code into a text string.

SIDXDisplayGetDisplay

status = **SIDXDisplayGetDisplay**(*SIDXDisplay*, *image_data*, *display*)

Parameters
- *SIDXDisplay* A handle value that references the SIDXDisplay context
- *image_data* The source image array for transformation.
-

Return value
- *status* Non-zero if the function failed, zero if it succeeded.

Transform the source image for display.

4.7 SIDXDevice

An object of this interface represents device specific operations in SIDX.

Reference

status = **SIDXDeviceActionDo**(*SIDXDevice*, *setting*, *command*)
status = **SIDXDeviceActionGetByItem**(*SIDXDevice*, *item*, *setting*)
status = **SIDXDeviceActionGetByName**(*SIDXDevice*, *name*, *setting*)
status = **SIDXDeviceActionGetCount**(*SIDXDevice*, *count*)
status = **SIDXDeviceActionGetName**(*SIDXDevice*, *item*, *name*)
status = **SIDXDeviceDriverGetDescription**(*SIDXDevice*, *description*)
status = **SIDXDeviceDriverGetName**(*SIDXDevice*, *name*)
status = **SIDXDeviceDriverGetType**(*SIDXDevice*, *type*)
status = **SIDXDeviceExtraBooleanGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraBooleanSet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraGetByItem**(*SIDXDevice*, *item*, *setting*)
status = **SIDXDeviceExtraGetByName**(*SIDXDevice*, *name*, *setting*)
status = **SIDXDeviceExtraGetCount**(*SIDXDevice*, *count*)
status = **SIDXDeviceExtraGetLabel**(*SIDXDevice*, *setting*, *label*)
status = **SIDXDeviceExtraGetName**(*SIDXDevice*, *item*, *name*)
status = **SIDXDeviceExtraGetType**(*SIDXDevice*, *setting*, *type*)
status = **SIDXDeviceExtraGetUnit**(*SIDXDevice*, *setting*, *unit*)
status = **SIDXDeviceExtraGetValueLocal**(*SIDXDevice*, *setting*, *description*)
status = **SIDXDeviceExtraIntegerGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraIntegerGetRange**(*SIDXDevice*, *setting*, *minimum*, *maximum*)
status = **SIDXDeviceExtraIntegerGetValue**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraIntegerSet**(*SIDXDevice*, *setting*, *integer*)
status = **SIDXDeviceExtraIsSettable**(*SIDXDevice*, *setting*, *settable*)
status = **SIDXDeviceExtraListGet**(*SIDXDevice*, *setting*, *item*)
status = **SIDXDeviceExtraListGetCount**(*SIDXDevice*, *setting*, *count*)
status = **SIDXDeviceExtraListGetEntry**(*SIDXDevice*, *setting*, *item*, *entry*)
status = **SIDXDeviceExtraListGetLocal**(*SIDXDevice*, *setting*, *item*, *description*)
status = **SIDXDeviceExtraListGetValue**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraListSet**(*SIDXDevice*, *setting*, *item*)
status = **SIDXDeviceExtraListSetValue**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraRealGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraRealGetRange**(*SIDXDevice*, *setting*, *minimum*, *maximum*)
status = **SIDXDeviceExtraRealGetValue**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraRealSet**(*SIDXDevice*, *setting*, *real*)
status = **SIDXDeviceExtraSequenceGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraSequenceGetSize**(*SIDXDevice*, *setting*, *size*)
status = **SIDXDeviceExtraSequenceSet**(*SIDXDevice*, *setting*, *value*)

status = **SIDXDeviceExtraStringGet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceExtraStringSet**(*SIDXDevice*, *setting*, *value*)
status = **SIDXDeviceGetDescription**(*SIDXDevice*, *description*)
status = **SIDXDeviceGetLabel**(*SIDXDevice*, *label*)
status = **SIDXDeviceGetName**(*SIDXDevice*, *name*)
status = **SIDXDevicePortAnalogGetRange**(*SIDXDevice*, *port*, *minimum*, *maximum*)
status = **SIDXDevicePortAnalogRead**(*SIDXDevice*, *port*, *voltage*)
status = **SIDXDevicePortAnalogWrite**(*SIDXDevice*, *port*, *voltage*)
status = **SIDXDevicePortBitRead**(*SIDXDevice*, *port*, *asserted*)
status = **SIDXDevicePortBitWrite**(*SIDXDevice*, *port*, *asserted*)
status = **SIDXDevicePortDigitalRead**(*SIDXDevice*, *port*, *value*)
status = **SIDXDevicePortDigitalWrite**(*SIDXDevice*, *port*, *value*)
status = **SIDXDevicePortGetCount**(*SIDXDevice*, *count*)
status = **SIDXDevicePortGetType**(*SIDXDevice*, *port*, *type*)

Methods

SIDXDeviceActionDo

status = **SIDXDeviceActionDo**(*SIDXDevice*, *setting*, *command*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the handle for the action.
- *command* A text string to be passed to the action. The meaning of the text string is action-specific. For many actions, the text string is not used and may be empty.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Perform a device-specific action.

SIDXDeviceActionGetByItem

status = **SIDXDeviceActionGetByItem**(*SIDXDevice*, *item*, *setting*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *item* An integer item of the action within the actions available for the camera. The first action available for the camera has item zero.
- *setting* A variable to receive an integer value representing the action setting value corresponding to the name. The handle value is zero if the item does not correspond to an action for the specific camera.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a handle for a device-specific action, based on the item of the action.

SIDXDeviceActionGetByName

status = **SIDXDeviceActionGetByName**(*SIDXDevice*, *name*, *setting*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *name* A text string containing the action name.
- *setting* A variable to receive an integer value representing the action setting value corresponding to the name. The handle value is zero if the name does not correspond to an action for the specific camera.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a handle for a device-specific action, based on the name of the action.

SIDXDeviceActionGetCount

status = **SIDXDeviceActionGetCount**(*SIDXDevice*, *count*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *count* A variable to receive an integer value representing the total count of device-specific actions. This value may be zero if the camera has no device-specific actions.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the total count of device-specific actions.

SIDXDeviceActionGetName

status = **SIDXDeviceActionGetName**(*SIDXDevice*, *item*, *name*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *item* An integer item of the action within the actions available for the camera. The first action available for the camera has item zero.
- *name* A variable to receive a text string representing the action name.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the name of a device-specific action, based on the item of the action. The name is consistent over time, so the same device-specific action always has the same name for a specific camera vendor. The name does not change across sessions.

SIDXDeviceDriverGetDescription

status = **SIDXDeviceDriverGetDescription**(*SIDXDevice*, *description*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *description* A variable to receive a text string description representing the driver. This description is intended for reporting to the user.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the driver description. The description text string is driver specific information. For example, it could contain the version number. The driver description may be useful for identifying the driver if you are having trouble with the driver.

SIDXDeviceDriverGetName

status = **SIDXDeviceDriverGetName**(*SIDXDevice*, *name*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *name* A variable to receive a text string name representing the driver.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the driver name. Typically the name is a text string that identifies the vendor and if necessary the specific driver. For example, if the vendor 'Acme' has two drives, 'Standard' and 'Plus', the driver name might be 'Acme Standard' or 'Acme Plus'.

SIDXDeviceDriverGetType

status = **SIDXDeviceDriverGetType**(*SIDXDevice*, *type*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *type* A variable to receive a value describing the device vendor and device driver type.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the device vendor and device driver.

SIDXDeviceExtraBooleanGet

status = **SIDXDeviceExtraBooleanGet**(*SIDXDevice*, *setting*, *value*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *value* A variable to receive a boolean value representing the current setting.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the value of a device-specific setting. This operation is valid only if the device-specific setting is a boolean type.

SIDXDeviceExtraBooleanSet

status = **SIDXDeviceExtraBooleanSet**(*SIDXDevice*, *setting*, *value*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *value* A boolean value to set.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the device-specific setting to the specified boolean value. This operation is valid only if the device-specific setting is a boolean type.

SIDXDeviceExtraGetByItem

status = **SIDXDeviceExtraGetByItem**(*SIDXDevice*, *item*, *setting*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *item* An integer value containing the item of the device-specific setting. The first setting has item zero.
- *setting* A variable to receive an integer value representing the setting value corresponding to the name. The value is zero if the item does not correspond to a setting for the specific camera.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a handle for a device-specific setting, based on the item of the setting.

SIDXDeviceExtraGetByName

status = **SIDXDeviceExtraGetByName**(*SIDXDevice*, *name*, *setting*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *name* A text string containing the setting name.
- *setting* A variable to receive an integer value representing the setting value corresponding to the name. The value is zero if the name does not correspond to a setting for the specific camera.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a handle for a device-specific setting, based on the name of the setting.

SIDXDeviceExtraGetCount

status = **SIDXDeviceExtraGetCount**(*SIDXDevice*, *count*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *count* A variable to receive an integer value representing the total count of device-specific settings. This value may be zero if the camera has no device-specific settings.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the total count of device-specific settings.

SIDXDeviceExtraGetLabel

status = **SIDXDeviceExtraGetLabel**(*SIDXDevice*, *setting*, *label*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *label* A variable to receive a text string representing the setting label, as a text string. If the handle is valid, the label will never be empty. If the handle is not valid, the label will be null.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a label for the device-specific setting as a text string. Typical label text strings are "Trigger edge" or "Serial shift time". These text strings are intended for display to a user.

SIDXDeviceExtraGetName

status = **SIDXDeviceExtraGetName**(*SIDXDevice*, *item*, *name*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *item* An integer value containing the item of the device-specific setting. The first setting has item zero.
- *name* A variable to receive a text string representing the setting name.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the name of a device-specific setting, based on the item of the setting. The name is consistent over time, so the same device-specific setting always has the same name for a specific camera vendor. The name does not change across sessions.

SIDXDeviceExtraGetType

status = **SIDXDeviceExtraGetType**(*SIDXDevice*, *setting*, *type*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *type* A variable to receive a value describing the type of the device-specific setting. If the handle is not valid, the setting type is none.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the type of a device-specific setting.

SIDXDeviceExtraGetUnit

status = **SIDXDeviceExtraGetUnit**(*SIDXDevice*, *setting*, *unit*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *unit* A variable to receive a text string representing the setting unit, as a text string. The text string will be empty if the setting has no units. The text string will be null if the handle is not valid.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the unit for the device-specific setting as a text string. Typical unit text strings are "ms" or "kHz". These unit text strings are intended for display to a user. Some device-specific settings may not have any units, the string will be empty. The setting units are independent of the setting value.

SIDXDeviceExtraGetValueLocal

status = **SIDXDeviceExtraGetValueLocal**(*SIDXDevice*, *setting*, *description*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *description* A variable to receive a text string description of the current value of the setting.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current value of a device-specific setting as a text string.

SIDXDeviceExtraIntegerGet

status = **SIDXDeviceExtraIntegerGet**(*SIDXDevice*, *setting*, *value*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *value* A variable to receive an integer value representing the setting value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the value of a device-specific setting. This operation is valid only if the device-specific setting is an integer type.

SIDXDeviceExtraIntegerGetRange

status = **SIDXDeviceExtraIntegerGetRange**(*SIDXDevice*, *setting*, *minimum*, *maximum*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *minimum* A variable to receive an integer value representing the minimum limit.
- *maximum* A variable to receive an integer value representing the maximum limit.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the minimum and maximum value for a device-specific setting. This operation is valid only if the device-specific setting is an integer type.

SIDXDeviceExtraIntegerGetValue

status = **SIDXDeviceExtraIntegerGetValue**(*SIDXDevice*, *setting*, *value*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *value* A variable to receive an integer value representing the setting value used by the camera.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the value used by the camera for a device-specific setting. This operation is valid only if the device-specific setting is an integer type.

SIDXDeviceExtraIntegerSet

status = **SIDXDeviceExtraIntegerSet**(*SIDXDevice*, *setting*, *integer*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *value* The integer value to set. The value must be valid for the setting.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the device-specific setting to the specified integer value. This operation is valid only if the device-specific setting is an integer type.

SIDXDeviceExtraIsSettable

status = **SIDXDeviceExtraIsSettable**(*SIDXDevice*, *setting*, *settable*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *settable* A variable to receive a boolean value to determine whether the setting parameter is settable.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Determine is the camera specific setting can be set, based on the returned setting handle.

SIDXDeviceExtraListGet

status = **SIDXDeviceExtraListGet**(*SIDXDevice*, *setting*, *item*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *item* A variable to receive an integer value representing the item of the entry in the setting list.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the current item value of the device-specific setting. This operation is valid only if the device-specific setting is a list type.

SIDXDeviceExtraListGetCount

status = **SIDXDeviceExtraListGetCount**(*SIDXDevice*, *setting*, *count*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *count* A variable to receive an integer value count containing the number of list entries for the device-specific setting. If the handle is not valid, or if the setting is not a 'list', this value is zero.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the count of available item values for a given device-specific setting. This operation is valid only if the setting type is 'list', the value returned is the total number of settings in the list.

SIDXDeviceExtraListGetEntry

status = **SIDXDeviceExtraListGetEntry**(*SIDXDevice*, *setting*, *item*, *entry*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *item* An integer item of the setting within the setting list. The first item in the setting list has item zero.
- *entry* A variable to receive a real (floating-point) value representing the entry associated with the specific item.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the value associated with a specific device-specific setting item.

SIDXDeviceExtraListGetLocal

status = **SIDXDeviceExtraListGetLocal**(*SIDXDevice*, *setting*, *item*, *description*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *item* An integer item of the setting within the setting list. The first item in the setting list has item zero.
- *description* A variable to receive a text string description of the specific item.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a description of a specific entry for a device-specific setting. This operation is valid only if the device-specific setting is a list type.

SIDXDeviceExtraListGetValue

status = **SIDXDeviceExtraListGetValue**(*SIDXDevice*, *setting*, *value*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *value* A variable to receive a real (floating-point) value representing the setting. The value is device-specific, the same value may represent a different setting on different cameras. However, the value should be consistent, so it should be possible restore the camera to the same setting later by setting the camera to this value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the value of a device-specific setting. This operation is valid only if the device-specific setting is a list type.

SIDXDeviceExtraListSet

status = **SIDXDeviceExtraListSet**(*SIDXDevice*, *setting*, *item*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *item* An integer item of the setting within the setting list. The first entry in the setting list has item zero.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the device-specific setting to the specified list item. This operation is valid only if the device-specific setting is a list type.

SIDXDeviceExtraListSetValue

status = **SIDXDeviceExtraListSetValue**(*SIDXDevice*, *setting*, *value*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *value* A real (flaoting-point) value to set. The value must be valid for the setting. The value is device-specific.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the device-specific setting to the specified integer value. This operation is valid only if the device-specific setting is a list type.

SIDXDeviceExtraRealGet

status = **SIDXDeviceExtraRealGet**(*SIDXDevice*, *setting*, *value*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *value* A variable to receive a real (floating-point) value representing the current setting value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the value of a device-specific setting. This operation is valid only if the device-specific setting is a real type.

SIDXDeviceExtraRealGetRange

status = **SIDXDeviceExtraRealGetRange**(*SIDXDevice*, *setting*, *minimum*, *maximum*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *minimum* A variable to receive a real (floating point) value representing the minimum limit.
- *maximum* A variable to receive a real (floating point) value representing the maximum limit.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the minimum and maximum value for a device-specific setting. This operation is valid only if the device-specific setting is a real type.

SIDXDeviceExtraRealGetValue

status = **SIDXDeviceExtraRealGetValue**(*SIDXDevice*, *setting*, *value*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *value* A variable to receive a real (floating-point) value representing the setting value used by the camera.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the value used by the device for a device-specific setting. This operation is valid only if the device-specific setting is a real type.

SIDXDeviceExtraRealSet

status = **SIDXDeviceExtraRealSet**(*SIDXDevice*, *setting*, *real*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *value* The real (floating-point) value to set.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the device-specific setting to the specified real value. This operation is valid only if the device-specific setting is a real type.

SIDXDeviceExtraSequenceGet

status = **SIDXDeviceExtraSequenceGet**(*SIDXDevice*, *setting*, *value*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *value* A variable to receive an array of integer value representing the current setting.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the value of a device-specific setting. This operation is valid only if the device-specific setting is an array of integer type.

SIDXDeviceExtraSequenceGetSize

status = **SIDXDeviceExtraSequenceGetSize**(*SIDXDevice*, *setting*, *size*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *size* A variable to receive an integer value representing the size of the array length, measured in units of integer, for the setting.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the array size for a device-specific setting. This operation is valid only if the device-specific setting is an array of integer type.

SIDXDeviceExtraSequenceSet

status = **SIDXDeviceExtraSequenceSet**(*SIDXDevice*, *setting*, *value*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *value* An array of integer value to set.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the device-specific setting to the specified array of integer value. This operation is valid only if the device-specific setting is an array of integer type.

SIDXDeviceExtraStringGet

status = **SIDXDeviceExtraStringGet**(*SIDXDevice*, *setting*, *value*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *value* A variable to receive a text string value representing the current setting.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the value of a device-specific setting. This operation is valid only if the device-specific setting is a text string type.

SIDXDeviceExtraStringSet

status = **SIDXDeviceExtraStringSet**(*SIDXDevice*, *setting*, *value*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *setting* An integer value containing the setting identifier.
- *value* A text string value to set.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Set the device-specific setting to the specified text string value. This operation is valid only if the device-specific setting is a text string type.

SIDXDeviceGetDescription

status = **SIDXDeviceGetDescription**(*SIDXDevice*, *description*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *description* A variable to receive a text string description of the device.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a description of the device. The description contains information that may be helpful for you to identify the device to the vendor.

SIDXDeviceGetLabel

status = **SIDXDeviceGetLabel**(*SIDXDevice*, *label*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *label* A variable to receive a text string label representing the device.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain a human readable label representing the device.

SIDXDeviceGetName

status = **SIDXDeviceGetName**(*SIDXDevice*, *name*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *name* A variable to receive a text string name representing the device.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the name of the device.

SIDXDevicePortAnalogGetRange

status = **SIDXDevicePortAnalogGetRange**(*SIDXDevice*, *port*, *minimum*, *maximum*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *port* An integer value representing the index of the port.
- *minimum* A variable to receive a real (floating point) value representing the minimum limit.
- *maximum* A variable to receive a real (floating point) value representing the maximum limit.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the minimum and maximum analog output voltage values.

SIDXDevicePortAnalogRead

status = **SIDXDevicePortAnalogRead**(*SIDXDevice*, *port*, *voltage*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *port* An integer value representing the index of the port.
- *voltage* A variable to receive a real (floating-point) value representing the input voltage.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtains the voltage on the specified port.

SIDXDevicePortAnalogWrite

status = **SIDXDevicePortAnalogWrite**(*SIDXDevice*, *port*, *voltage*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *port* An integer value representing the index of the port.
- *voltage* A real (floating-point) value representing the output voltage.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the voltage as a fraction of maximum possible on the specified port.

SIDXDevicePortBitRead

status = **SIDXDevicePortBitRead**(*SIDXDevice*, *port*, *asserted*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *port* An integer value representing the index of the port.
- *asserted* A variable to receive a boolean value, true if asserted, false otherwise.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the status of the digital input bit specified by the index parameter, as retrieved by the most recent successful DigitalQuery.

SIDXDevicePortBitWrite

status = **SIDXDevicePortBitWrite**(*SIDXDevice*, *port*, *asserted*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *port* An integer value representing the index of the port.
- *asserted* A boolean value, true if the specified output bit is asserted. False if the specified output bit is deasserted.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the digital output bit specified by the index parameter to the value specified by the asserted parameter

SIDXDevicePortDigitalRead

status = **SIDXDevicePortDigitalRead**(*SIDXDevice*, *port*, *value*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *port* An integer value representing the index of the port
- *value* A variable to receive an integer value representing the digital input value.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Gets the digital input value.

SIDXDevicePortDigitalWrite

status = **SIDXDevicePortDigitalWrite**(*SIDXDevice*, *port*, *value*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *port* An integer value representing the index of the port
- *value* An integer value to be written to the digital outputs.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Sets the digital output bits to the values specified by the data parameter

SIDXDevicePortGetCount

status = **SIDXDevicePortGetCount**(*SIDXDevice*, *count*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *count* A variable to receive an integer value representing the total count of I/O ports.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtains the I/O port count.

SIDXDevicePortGetType

status = **SIDXDevicePortGetType**(*SIDXDevice*, *port*, *type*)

Parameters

- *SIDXDevice* A handle value that references the SIDXDevice context
- *An* integer value representing the port index.
- *type* A variable to receive a value corresponding to the port type.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtains the available port type for a given port index. The type determines the I/O functions that are valid for the port to read and write.

4.8 SIDXGeometry

An object of this interface represents an image geometry in SIDX.

Reference

status = **SIDXGeometryChannelGetDepth**(*SIDXGeometry*, *depth*)
status = **SIDXGeometryImageGetSize**(*SIDXGeometry*, *size*)
status = **SIDXGeometryImageGetType**(*SIDXGeometry*, *type*)
status = **SIDXGeometryPixelGetCount**(*SIDXGeometry*, *x*, *y*)
status = **SIDXGeometryPixelSpacingGet**(*SIDXGeometry*, *x*, *y*)

Methods

SIDXGeometryChannelGetDepth

status = **SIDXGeometryChannelGetDepth**(*SIDXGeometry*, *depth*)

Parameters
- *SIDXGeometry* A handle value that references the SIDXGeometry context
- *depth* A variable to receive an integer value representing the channel depth.

Return value
- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the pixel depth of one channel from the image data. For grayscale image, there is one channel. For RGB image, there are three channels which represent red, green and blue respectively.

SIDXGeometryImageGetSize

status = **SIDXGeometryImageGetSize**(*SIDXGeometry*, *size*)

Parameters
- *SIDXGeometry* A handle value that references the SIDXGeometry context
- *size* A variable to receive an integer value representing the size of the data in an image, measured in bytes.

Return value
- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the data size of an image. The data size of an image is the size of the data in an image, excluding any padding.

SIDXGeometryImageGetType

status = **SIDXGeometryImageGetType**(*SIDXGeometry*, *type*)

Parameters

- *SIDXGeometry* A handle value that references the SIDXGeometry context
- *type* A variable to receive a value describing the type of source image.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the image type of the source image.

SIDXGeometryPixelGetCount

status = **SIDXGeometryPixelGetCount**(*SIDXGeometry*, *x*, *y*)

Parameters

- *SIDXGeometry* A handle value that references the SIDXGeometry context
- *x* A variable to receive an integer value representing the pixel count on the x axis.
- *y* A variable to receive an integer value representing the pixel count on the y axis.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the x and y pixel count for images.

SIDXGeometryPixelSpacingGet

status = **SIDXGeometryPixelSpacingGet**(*SIDXGeometry*, *x*, *y*)

Parameters

- *SIDXGeometry* A handle value that references the SIDXGeometry context
- *x* A variable to receive a real (floating point) value representing the horizontal pixel spacing, measured in meters.
- *y* A variable to receive a real (floating point) value representing the vertical pixel spacing, measured in meters.

Return value

- *status* Non-zero if the function failed, zero if it succeeded.

Obtain the pixel size in x and y.

4.9 Constants

SIDXCoolingControl

A value of this type represents a temperature control capability.

Value	Name
0	SIDX_COOLING_CONTROL_NONE
1	SIDX_COOLING_CONTROL_GET_ONLY
2	SIDX_COOLING_CONTROL_GET_SET

Descriptions

SIDX_COOLING_CONTROL_NONE

The camera does not provide any temperature information, and does not provide any temperature control.

SIDX_COOLING_CONTROL_GET_ONLY

The camera provides the sensor temperature, but does not provide control of the sensor temperature.

SIDX_COOLING_CONTROL_GET_SET

The camera provides the sensor temperature, and also provides control of the sensor temperature.

SIDXDriverType

A value of this type represents a device driver. The text "*not currently supported*" indicates that the driver is not currently supported by SIDX. Continued updates to SIDX will include additional driver support.

Value	Name
0	SIDX_DRIVER_TYPE_ANDOR_TECHNOLOGY
1	SIDX_DRIVER_TYPE_BRUXTON
2	SIDX_DRIVER_TYPE_COOKE_PCO_SENSICAM
3	SIDX_DRIVER_TYPE_DVC
4	SIDX_DRIVER_TYPE_HAMAMATSU
5	SIDX_DRIVER_TYPE_QIMAGING
6	SIDX_DRIVER_TYPE_PHOTOMETRICS_PRINCETON_INSTRUMENTS_PVCAM
7	SIDX_DRIVER_TYPE_SCIMEASURE
8	SIDX_DRIVER_TYPE_COOKE_PCO_DOT_CAMERA
9	SIDX_DRIVER_TYPE_COOKE_PCO_PIXELFLY
10	SIDX_DRIVER_TYPE_JENOPTIK
11	SIDX_DRIVER_TYPE_AVT
12	SIDX_DRIVER_TYPE_APPLIED_SCIENTIFIC_INSTRUMENTATION
13	SIDX_DRIVER_TYPE_LUDL_ELECTRONIC_PRODUCTS
14	SIDX_DRIVER_TYPE_PRIOR_SCIENTIFIC

Descriptions

SIDX_DRIVER_TYPE_ANDOR_TECHNOLOGY

Andor Technology *not currently supported*

SIDX_DRIVER_TYPE_BRUXTON

Bruxton Corporation

SIDX_DRIVER_TYPE_COOKE_PCO_SENSICAM

PCO sensicam *not currently supported*

SIDX_DRIVER_TYPE_DVC

DVC *not currently supported*

SIDX_DRIVER_TYPE_HAMAMATSU

Hamamatsu *not currently supported*

SIDX_DRIVER_TYPE_QIMAGING

QImaging *not currently supported*

SIDX_DRIVER_TYPE_PHOTOMETRICS_PRINCETON_INSTRUMENTS_PVCAM

Photometrics Princeton Instruments PVCAM *not currently supported*

SIDX_DRIVER_TYPE_SCIMEASURE

SciMeasure Analytical Systems

SIDX_DRIVER_TYPE_COOKE_PCO_DOT_CAMERA

PCO .camera *not currently supported*

SIDX_DRIVER_TYPE_COOKE_PCO_PIXELFLY

PCO pixelfly

SIDX_DRIVER_TYPE_JENOPTIK

Jenoptik

SIDX_DRIVER_TYPE_AVT

AVT

SIDX_DRIVER_TYPE_APPLIED_SCIENTIFIC_INSTRUMENTATION

Applied Scientific Instrumentation

SIDX_DRIVER_TYPE_LUDL_ELECTRONIC_PRODUCTS

Ludl Electronic Products *not currently supported*

SIDX_DRIVER_TYPE_PRIOR_SCIENTIFIC

Prior Scientific *not currently supported*

SIDXImageType

A value of this type represents either a source image type or a target image type.

Value	Name
0	SIDX_IMAGE_TYPE_GRAYSCALE16
1	SIDX_IMAGE_TYPE_RBG24
2	SIDX_IMAGE_TYPE_RBG48

Descriptions

SIDX_IMAGE_TYPE_GRAYSCALE16

Black and white source image type.

SIDX_IMAGE_TYPE_RBG24

Target image type.

SIDX_IMAGE_TYPE_RBG48

Color source image type.

SIDXPortType

A value of this type represents the device port.

Value	Name
0	SIDX_PORT_TYPE_ANALOG_IN
1	SIDX_PORT_TYPE_ANALOG_OUT
2	SIDX_PORT_TYPE_BIT_IN
3	SIDX_PORT_TYPE_BIT_OUT
4	SIDX_PORT_TYPE_DIGITAL_IN
5	SIDX_PORT_TYPE_DIGITAL_OUT
6	SIDX_PORT_TYPE_COUNT

Descriptions

SIDX_PORT_TYPE_ANALOG_IN

Analog input

SIDX_PORT_TYPE_ANALOG_OUT

Analog output

SIDX_PORT_TYPE_BIT_IN

Bit input

SIDX_PORT_TYPE_BIT_OUT

Bit output

SIDX_PORT_TYPE_DIGITAL_IN

Digital input

SIDX_PORT_TYPE_DIGITAL_OUT

Digital output

SIDX_PORT_TYPE_COUNT

The total number of possible device ports.

SIDXSettingType

A value of this type represents a setting condition.

Value	Name
0	SIDX_SETTING_TYPE_BOOLEAN
1	SIDX_SETTING_TYPE_INTEGER
2	SIDX_SETTING_TYPE_LIST
3	SIDX_SETTING_TYPE_NONE
4	SIDX_SETTING_TYPE_REAL
5	SIDX_SETTING_TYPE_SEQUENCE
6	SIDX_SETTING_TYPE_STRING

Descriptions

SIDX_SETTING_TYPE_BOOLEAN

The setting is one of the values true or false.

SIDX_SETTING_TYPE_INTEGER

The setting value is selected from a range of integer values. The range is identified by a minimum value and a maximum value.

SIDX_SETTING_TYPE_LIST

The setting value is selected from a small set (list) of values. The list is specified by a count of possible values. The value item range is 0 (zero) to one less than the count of possible values.

SIDX_SETTING_TYPE_NONE

No setting available.

SIDX_SETTING_TYPE_REAL

The setting value is selected from a range of real (floating point) values. The range is identified by a minimum value and a maximum value.

SIDX_SETTING_TYPE_SEQUENCE

The setting is an array of integer values.

SIDX_SETTING_TYPE_STRING

The setting is a text string.

SIDXShutterMode

A value of this type represents a shutter control mode.

Value	Name
0	SIDX_SHUTTER_MODE_OPEN_DURING_EXPOSURE
1	SIDX_SHUTTER_MODE_OPEN_DURING_SEQUENCE
2	SIDX_SHUTTER_MODE_OPEN_DURING_ENABLE
3	SIDX_SHUTTER_MODE_OPEN
4	SIDX_SHUTTER_MODE_CLOSE
5	SIDX_SHUTTER_MODE_COUNT

Descriptions

SIDX_SHUTTER_MODE_OPEN_DURING_EXPOSURE

The shutter is open during each individual exposure. The shutter closes between exposures.

SIDX_SHUTTER_MODE_OPEN_DURING_SEQUENCE

The shutter is open during an exposure sequence. The shutter closes when no sequence is being acquired. If the sequence is triggered, the shutter opens only when the trigger occurs.

SIDX_SHUTTER_MODE_OPEN_DURING_ENABLE

The shutter is open while image acquisition is enabled. The shutter remains open during the acquisition sequence. If the sequence is triggered, the shutter opens as soon as acquisition is enabled, so the shutter may be open before the trigger occurs. The shutter closes when image acquisition is not enabled.

SIDX_SHUTTER_MODE_OPEN

The shutter is always open.

SIDX_SHUTTER_MODE_CLOSE

The shutter is always closed.

SIDX_SHUTTER_MODE_COUNT

The total number of possible shutter modes.

SIDXSignalActiveMode

A value of this type represents the trigger signal mode for acquisition.

Value	Name
0	SIDX_SIGNAL_ACTIVE_MODE_EDGE_RISING
1	SIDX_SIGNAL_ACTIVE_MODE_EDGE_FALLING
2	SIDX_SIGNAL_ACTIVE_MODE_EDGE_ANY
3	SIDX_SIGNAL_ACTIVE_MODE_LEVEL_HIGH
4	SIDX_SIGNAL_ACTIVE_MODE_LEVEL_LOW
5	SIDX_SIGNAL_ACTIVE_MODE_COUNT

Descriptions

SIDX_SIGNAL_ACTIVE_MODE_EDGE_RISING

SIDX_SIGNAL_ACTIVE_MODE_EDGE_FALLING

SIDX_SIGNAL_ACTIVE_MODE_EDGE_ANY

SIDX_SIGNAL_ACTIVE_MODE_LEVEL_HIGH

SIDX_SIGNAL_ACTIVE_MODE_LEVEL_LOW

SIDX_SIGNAL_ACTIVE_MODE_COUNT

The total number of possible trigger signal modes.

SIDXStatus

A value of this type represents a status code.

Value	Name
0	SIDX_STATUS_SUCCESS
1	SIDX_STATUS_PARAMETER_INVALID
2	SIDX_STATUS_RESOURCES_INSUFFICIENT
3	SIDX_STATUS_DEVICE_ERROR
4	SIDX_STATUS_DEVICE_NOT_FOUND
5	SIDX_STATUS_CONFIGURATION_ERROR
6	SIDX_STATUS_INTERNAL_ERROR
7	SIDX_STATUS_OPERATION_INVALID

Descriptions

SIDX_STATUS_SUCCESS

No errors.

SIDX_STATUS_PARAMETER_INVALID

The parameter is invalid.

SIDX_STATUS_RESOURCES_INSUFFICIENT

The memory resource is insufficient.

SIDX_STATUS_DEVICE_ERROR

An error has been detected in the hardware.

SIDX_STATUS_DEVICE_NOT_FOUND

The hardware device is not found.

SIDX_STATUS_CONFIGURATION_ERROR

The current hardware configuration is not supported.

SIDX_STATUS_INTERNAL_ERROR

Unanticipated error. Please report.

SIDX_STATUS_OPERATION_INVALID

The operation is not valid in the current conditions.

SIDXTriggerInMode

A value of this type represents the trigger input control mode for acquisition.

Value	Name
0	SIDX_TRIGGER_IN_MODE_ALWAYS
1	SIDX_TRIGGER_IN_MODE_EXPOSURE_START
2	SIDX_TRIGGER_IN_MODE_EXPOSURE_DURATION
3	SIDX_TRIGGER_IN_MODE_SEQUENCE_START
4	SIDX_TRIGGER_IN_MODE_COUNT

Descriptions

SIDX_TRIGGER_IN_MODE_ALWAYS

The trigger is always asserted. An acquisition will start immediately

SIDX_TRIGGER_IN_MODE_EXPOSURE_START

The trigger input starts a single exposure. The exposure duration is set by the exposure interval.

SIDX_TRIGGER_IN_MODE_EXPOSURE_DURATION

The trigger input gates a single exposure. The exposure duration is set by the trigger, the camera exposes the image as long as the the trigger is asserted. The camera terminates the exposure when the trigger is deasserted. This is also called 'bulb' mode.

SIDX_TRIGGER_IN_MODE_SEQUENCE_START

The trigger input starts an acquisition sequence. The trigger input is asserted only once for the entire sequence.

SIDX_TRIGGER_IN_MODE_COUNT

The total number of possible trigger input modes.