Duplex/Simplex

Mobile Scanner Development Kit

Document Capture Technologies Inc.

Duplex/Simplex

Mobile Scanner Development Kit

1798 Technology Drive, Suite 178 • San Jose CA 95110
Phone 408.436.9888 • Fax 408.436.6151
www.docucap.com
sdkresources@docucap.com

Table of Contents

Introduction

The DCT Duplex/Simplex Mobile Scanner Development Kit provides an application programming interface for controlling DCT scanners. This interface is implemented as a dynamically linked library (DLL) for the Microsoft Windows platform. Through this interface, application programs are able to easily perform scanner operations such as calibrating the scanner, detecting paper, checking button state, and scanning pages in various scan modes and resolutions. Control is provided for various parameters such as brightness, contrast, gamma, highlight and shadow, and custom lookup tables.

By handling the low-level operation of the scanner hardware, DCT's scanner API allows system integrators to shorten development time and focus on application-specific development.

The DCT API DLL supports Microsoft Windows 7, Vista, XP, 2000.

Package Contents

- Documentation (including this file)
- DCT drivers for one or more scanner models.
- Header and library files.
- Sample application source code
- A Nondisclosure and Confidential Agreement
- Two support incidents through email or one support incident through telephone.
- Optional extra incident support through Sales

For questions or comments related to this SDK, please send email to sdkresources@docucap.com.

Requirements

The DCT files are compatible with Microsoft Visual C/C++ 6.0, Microsoft Visual C/C++ .NET, and later compilers. Compilers by other vendors may work but are not tested or supported. Development with managed languages such as C# and Visual Basic .NET requires Microsoft Visual Studio 2005 or later.

The recommended development system is Microsoft Windows 7, Vista, or XP.

Scanner Installation

Included in the SDK is a Windows INF file used to install the device driver for your scanner. The driver must be installed before using the SDK. To install, follow these steps:

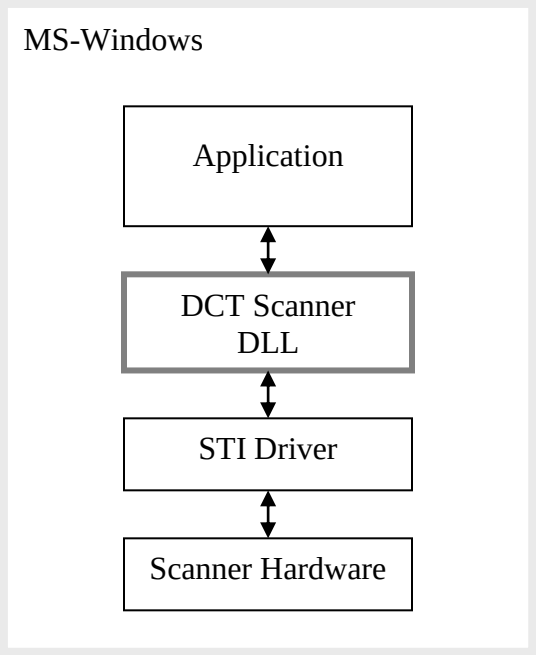
1. Plug the scanner into the computer's USB port. The MS-Windows New Hardware Wizard should appear.
2. Direct the New Hardware Wizard to the folder where the DCT driver and INF file for your scanner model is stored. Note that there are separate directories for the 32-bit and 64-bit versions of the driver.
3. A dialog may appear warning you that the driver has not passed Windows Logo certification. You can just click OK to dismiss this dialog.
4. When the New hardware Wizard has completed, the driver is installed and ready for use. You can immediately try one of the sample applications to confirm the scanner operation.

Technical Overview

The application programming interface is implemented within a single DLL file as a set of exported, C-callable functions. The application calls the API functions to control the scanner, scan images, and to read image data. The DCT DLL handles all low-level communication with the scanner hardware through the Windows Still Image (STI) Interface.

All scan parameters can be queried and set by means of "properties". The application can query the current scan parameters as well as the range of valid values for the parameters. This offers flexibility and allows the developer to avoid hard-coding the scanner's capabilities and limits within the application.

The DLL provides no user interface elements; it provides data only. The application should implement its own user interface if needed.



Including the DLL in Your Application

To build your application to work with the DCT DLL, do the following two steps.

1. Add the line

```
#include  "ScannerAPI.h"
```

at the top of each source file that makes use of the API. This file in turn includes the header "PlatTypes.h" which must also be in the include file path.

2. If you intend to use implicit linking, include the library file (such as `DPORT487.LIB` for the DocketPORT 487) as an additional dependency in the linker section of your project.

That will cause the DLL to be linked when the application is loaded. Alternatively, you can load the DLL at run-time (explicit linking) using the Windows function `LoadLibrary()`, passing the filename such as "`DPORT487.dll`" as a parameter (or whatever the library filename is for the particular scanner model).

The library DLL filename to load for some scanner models:

Scanner model	Library DLL filename
DocketPORT 487	`DPORT487.DLL`
DocketPORT 488	`DPORT488.DLL`
DocketPORT 467	`DPORT467.DLL`
DocketPORT 468	`DPORT468.DLL`
DocketPORT 667	`DPORT667.DLL`
DocketPORT 687	`DPORT687.DLL`

Using the API

Initializing the Interface

To use the scanner, you begin by calling the function `SI_OpenInterface()`. This initializes the interface and must be called before any other functions. The parameter passed to this function is a textual name of the scanner, which depends on the scanner model. (You can find the text name in the addendum document for your particular scanner model.)

```
#include "ScannerAPI.h"

SIResult result;

result = SI_OpenInterface("DocketPORT487");
```

The `SI_OpenInterface()` function also initializes the scanner hardware. If the return value is SIR_SUCCESS, then the interface was opened successfully and the scanner initialized. If the scanner is not connected, you'll get an error result. For other possible return values, see `SI_OpenInterface()` in the function reference.

When you're finished using the scanner, you should call `SI_CloseInterface()` to close the interface and clean up any resources.

Calibrate the Scanner

Normally you'll want the user to calibrate the scanner before doing the first scan. You can check to see if the scanner has been calibrated by calling `SI_IsCalibrated()`. This function determines whether or not the scanner has been calibrated by checking for the presence of a calibration data file on disk. See the `SI_Calibrate()` in the function reference for information about this file.

```
SIResult result;
SICalibrationState calState;

result = SI_IsCalibrated(&calState);

if (result == SIR_SUCCESS)
{
      if(calState == SI_FALSE)
      {
            // The scanner has not been calibrated.

            // <Here you would check the paper status to make
            //  sure the calibration target is inserted.>

            // Perform the calibration.
            result = SI_Calibrate(SI_CT_DEFAULT, NULL);

            if (result == SIR_SUCCESS)
            {
                  printf("Calibration completed successfully.\n");
            }
            else
            {
                  printf("There was an error during calibration.\n");
            }
      }
}

// Continue on to scan an image
```

If `SI_IsCalibrated()` returns a result of `SIR_NOT_CALIBRATED`, the scanner has not yet been calibrated. You can then call `SI_Calibrate()` to calibrate the scanner. The first parameter passed specifies the type of calibration target that will be inserted. At this time, the only valid parameter is `SI_CT_DEFAULT`. The `SI_Calibrate()` function will not return until the calibration is finished or there is an error. The second parameter is an optional pointer to a callback function. See the `SI_Calibrate()` function reference for more detail.

Setup Parameters for Scanning

The scan parameters are set prior to starting a scan by means of scanner *properties*. Properties describe the parameters of a scan, such as resolution, scan window, scan mode, and the brightness. Properties are much like TWAIN capabilities or WIA item properties. You can set one property at a time. The API functions to allow you to read and write scanner properties are:

```
SIResult SI_GetProperty(SIProperty* pProperty);
SIResult SI_SetProperty(SIProperty* pProperty);
```

Getting and setting scanner properties are explained in more detail in the section on Properties. In short, to find out the current setting of a property, you specify the property ID you want in the `SIProperty` structure and then pass it to `SI_GetProperty()`.

```
SIProperty  prop;
SIResult    result;

prop.propertyID = SIP_XRESOLUTION;        // initialize the ID

result = SI_GetProperty(&prop);           // Get the property

assert(result == SIR_SUCCESS);            // make sure it succeeded

int32 currentRes = prop.list.current.iVal;   // get the resolution value
```

You can also use `SI_GetProperty()` to tell you which values are valid for a given property.

To set a given property, you need to fill out four members of a `SIProperty` structure and then pass it to `SI_SetProperty()`.

The following example sets the X resolution property to 100 DPI.

```
SIProperty  prop;
SIResult    result;

// Initialize property fields
prop.propertyID = SIP_XRESOLUTION;  // Set the property ID for X resolution
prop.containerType = SICON_LIST;    // uses a list container
prop.itemType = SI_INT32;           // Set the item type to 32-bit integer
prop.list.current.iVal = 100;       // Set resolution to 100 DPI

// Set the property's current value
result = SI_SetProperty(&prop);
assert(result == SIR_SUCCESS);
```

If the current value you tried to set is not one of the valid values, you'll get an error result and that property will not be changed. Therefore, it's not possible to set invalid parameters. Before you actually start the scan, you'll know that all scan parameters are valid.

For more information on properties, and on reading and writing them, see the section on Properties.

Check for Paper

Before starting a scan, you'll normally want to check whether or not there is paper inserted. You can to this by calling the function `SI_GetPaperStatus()`. The state of the paper sensor is returned in the `SIPaperStatus` parameter.

The following example checks to see if paper is inserted. If not, it will display a message prompting the user to insert the paper and hit a key.

```
SIResult          result;
SIPaperStatus     paperStatus;

// Check if the paper is in.
result = SI_GetPaperStatus(&paperStatus);

while(paperStatus == SI_PS_PAPER_OUT)
{
      printf("Insert paper and press any key to begin the scan.\n");

      // Wait for a keystroke
      while(!_kbhit())
      {};

      ch = _getch();    // pull in the character.

      // Check if the paper is in.
      result = SI_GetPaperStatus(&paperStatus);

      if(result != SIR_SUCCESS)
      {
            printf("Error getting paper status.\n");
      }
};
```

Scan a page

Once all properties are set and there is paper in the scanner, you can begin the scan. To do this, call `SI_StartScan()`, which takes no parameters. The scan will use the currently set properties.

```
SIResult    result;

result = SI_StartScan();

assert(result == SIR_SUCCESS);
```

Once the scan has started, the page will begin to feed. You can then read image data using the `SI_ReadImageData()` function. This function lets you read out a number of lines.

In the following code example, `pLineBuffer` is a pointer to buffer allocated by the application to receive the image data. The size of this buffer can be determined by querying the property `SIP_LINE_WIDTH_IN_BYTES`. This will tell you the number of bytes that are return for

one line and thus how much buffer space you need to provide. This example requests at most 10 lines per call, so the buffer is large enough to hold 10 lines of data.

```
int32 linesRemaining = 1100;
int32 linesToRead;


while(linesRemaining > 0)
{

	linesToRead = (linesRemaining < 10) ? linesRemaining : 10;

	// Read the image data
	result = SI_ReadImageData(pLineBuffer, linesToRead, 0, &linesReturned);

	if((result != SIR_SUCCESS) && (result != SIR_ENDOFDATA))
	{
		// Either there was an error, or we've read all the data.
		linesRemaining = 0;
		break;
	}

	if(linesReturned > 0)
	{
		// <Here, save the data to file, or copy to another buffer. >

		linesRemaining -= linesReturned;
	}
	else
	{
		// Since 0 lines were returned, we've caught up to the scanner.
		// Wait a bit for more data.
		Sleep(10);
	}
};

// End the scan
SI_EndScan();
```

If you do not have the end-of-page detection feature enabled, then you can just read the number of lines and stop, since you already know the number of lines that will be scanned. The example uses the variable `linesRemaining` to read 1100 lines of data.

If you *do* have end-of-page detection enabled (by setting the `SIP_EOP_DETECT_ENABLED` property) then you won't know how many lines can be read. In this case you need to keep reading lines until `SI_ReadImageData()` returns a result of SIR_ENDOFDATA. This means that there are no lines left for the page.

When you are done reading image data, call `SI_EndScan()` to end the scan for this page. This can be called anytime after `SI_StartScan()` and you do not need to read all the image data. When `SI_EndScan()` is called, the paper feeding will stop.

Note that only certain API functions are allowed to be called between the `SI_StartScan()/SI_EndScan()` pair. These include:

```
SI_ReadImageData()
SI_GetPaperStatus()
SI_GetScannerStatus()
SI_GetButtonStatus()
```

Also, `SI_ReadImageData()` can only be called between the `SI_StartScan()/SI_EndScan()` pair.

Since `SI_GetPaperStatus()` can be called during a scan, it's possible to do your own end-of-page detection by polling the paper status during a scan. This method is less accurate, however, if spooling is enabled. You would also need to take into account the distance from the paper sensor to image sensor. This is already done automatically if you allow the DLL to handle end-of-page detection.

Feed the Paper out

When the scan is done, the paper may not be completely fed out of the scanner. This can depend on the paper path of the particular scanner model.

You can feed the paper out using the `SI_FeedPaperOut()` function. This will feed the paper until the paper sensor is cleared. The function will not return until the sensor is clear, or until some maximum feed length has been reach without the sensor becoming clear. `SI_FeedPaperOut()` will always feed the paper some minimum distance even if the sensor is clear at the start. This is to make sure the paper path is clear, since there is usually some distance after the sensor is clear until the paper path is clear. The distance depends on the scanner model.

```
// Feed the paper out of the scanner.
SI_FeedPaperOut();
```

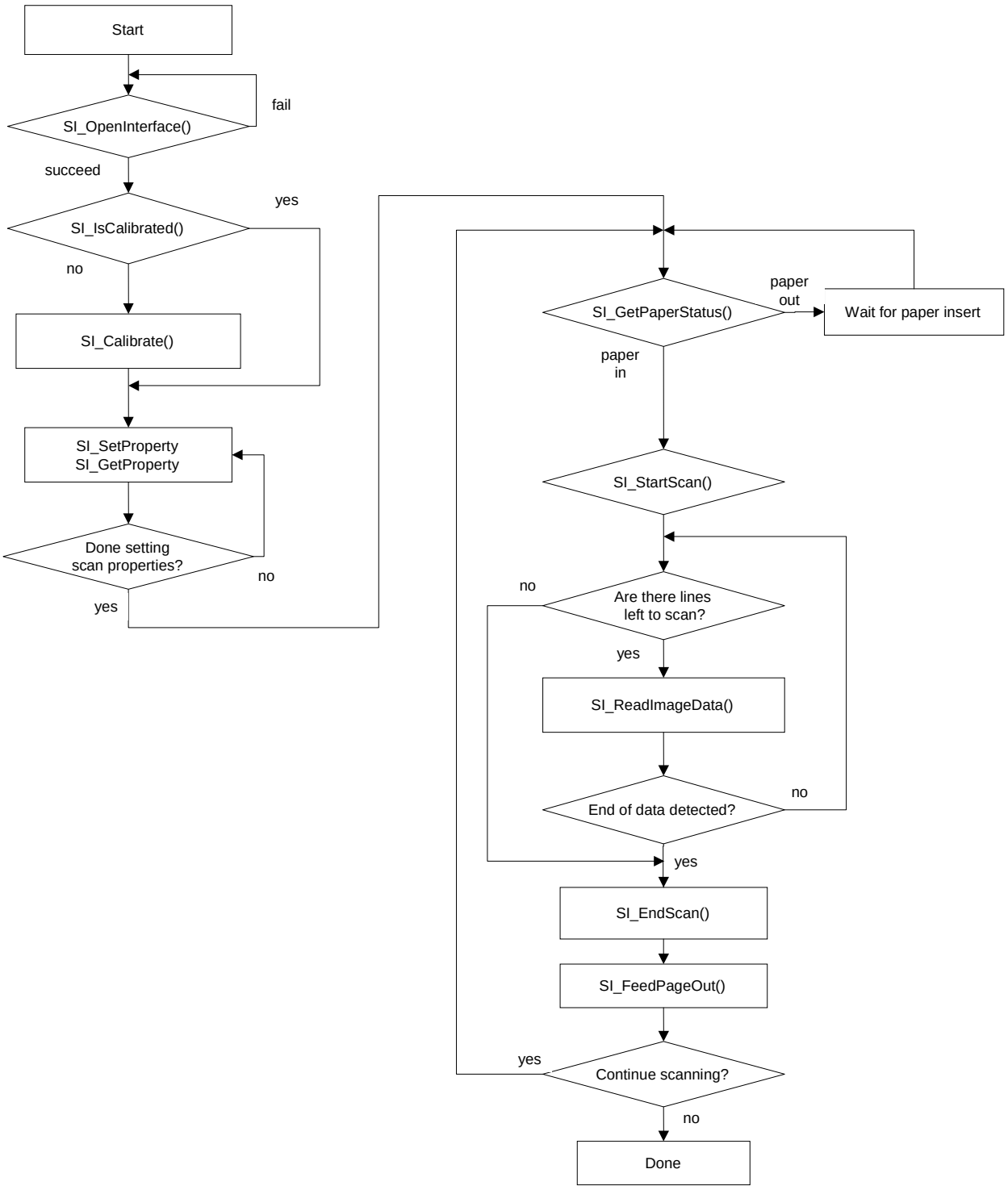
You can also feed paper arbitrarily without performing a scan using the `SI_Feed()` function. This is useful for feeding the paper out when the user cancels the scan. Before calling `SI_Feed()`, you need to set up properties that define the feed characteristics. You can set the

length of the feed and perhaps also the feed direction if that property is supported in the given scanner model. For more information on how to use `SI_Feed()`, refer to the function reference.

Close the Interface

When you're finished scanning, you must call SI_CloseInterface(). This will close the interface and clean up any resources. After calling this function, you can no longer call any API functions until SI_OpenInterface() is called again.

Typical Program Flow Diagram



Calling API Functions From Multiple Processes

The DCT DLL can be used with multiple processes in which a separate instance of the DLL is loaded for each process. At the lowest level, the DLL makes use of a globally unique mutex to synchronize communication with the scanner over the USB bus. Thus, separate processes will not conflict when calling API functions. Two processes could, for example, check the paper sensor status at the same time. Since the synchronization is global, the two processes could exist in separate user sessions. This also allows multiple processes to get and set scan properties separately before starting a scan. Scan properties are not sent to the scanner hardware until the scan is started. Therefore, if two processes are setting the resolution to different values, they will not conflict with each other.

However, only one process at a time can perform a scan (or a feed). Once one process calls **SI_StartScan()**, a second process cannot then call **SI_StartScan()** until the first process calls **SI_EndScan()** to end the scan process. In such a case, the second process would get a result of SIR_SCANNER_BUSY. This also applies to **SI_Feed**, **SI_FeedPageOut()** and **SI_Calibrate**.

The DLL is not thread-safe. If your application calls the DLL from more than one thread, you should ensure that only one thread at a time calls it.

About Calibration

Due to normal differences in the image sensor from one scanner to the next, each scanner must be calibrated. Calibration should be done before the first use, but it does not need to be done before each use. Typically users only need to re-calibrate every few months or when the image quality seems noticeably degraded. To learn how to start the calibration process from your application program, see the section Calibrate the Scanner.

DocketPORT drivers also allow users to calibrate from the Windows Control Panel. With the scanner plugged in, open the Windows Control Panel and double-click on the Scanners and Cameras applet. Right-click on the scanner icon and select Properties. On Windows XP, this opens the properties dialog. (There are slightly different ways to open the Properties dialog on Vista and later platforms.) Select the Advanced tab. From here you can start calibration. Insert the calibration target to the scanner in the direction indicated by the arrows and press the Calibrate button. Calibration should normally take about 20-30 seconds. You can also calibrate with a clean white sheet of paper if the calibration target is not available. Make sure the paper width extends to the full width of the scanner opening and is at least as long as the calibration target.

The result of calibration is a set of data, or a “profile”, corresponding to that scanner. The profile data is not saved in the scanner itself but in a file on the host computer’s hard drive. Therefore, if

you move the scanner from one computer to another, you will need to recalibrate the scanner. This file is named Calibration.dat and on Windows XP you can find it at:

```
C:\Documents and Settings\All Users\Application Data\Docucap\<scanner model>\Calibration.dat
```

…where *<scanner model>* is the name of the scanner model, such as "DocketPORT487". On Vista and later platforms:

```
C:\ProgramData\DocuCap\<scanner model>\Calibration.dat
```

User accounts without administrator privilege will not be able to calibrate the scanner because they will not have permission to overwrite the Calibration.dat file.

Debugging

When debugging your application, be aware that DocketPORT scanners incorporate a watchdog function as a safety feature. This watchdog function is designed to disable power to the motor when there has been no communication with the host computer for about 75 seconds. When stepping through your program in a debugger, you may call the SI_StartScan function to begin a scan but then stop on a breakpoint before calling SI_EndScan. If you pause at this breakpoint longer than 75 seconds, the watchdog timer will timeout and the scan will be aborted. Further calls to SI_ReadImageData, for example, will not cause the scan to continue. You can continue on to call SI_EndScan and then begin another scan.

Image Spooling

The DCT DLL incorporates an image spooler. The spooler runs in a separate thread and, once the scan is started, begins reading the image data from the scanner hardware and writing it to a spool file on disk. This allows the scan to continue without stopping even when the application does not read the image data in a timely manner.

Image spooling can be disabled if desired by setting a scan property. However, it's then the responsibility of the application to read image data quickly enough to prevent the scanner from pausing when its buffer becomes full. If the scanner pauses, it is not an error and the scan can continue normally once the application continues reading the data. But pausing can possibly cause slight artifacts in the image, and stopping and starting may be less pleasant from the user's perspective. If the scanner is a duplex scanner, then the spooler must be enabled in order to read both sides. If duplex is enabled and the spooler is not enabled, then only one side or the other can be read.

The name and location of the spool file depends on the user account and the operating system. It is a temporary file; the name and location are recommended by the operating system. On Windows systems, the file is typically stored in the user's temp folder with a path like

```
C:\Documents and Settings\<username>\Local Settings\Temp
```

Or if that can't be used (such as on Windows 98SE, for example), then the file will be stored in `C:\Windows\temp`. The file is created when the application calls **SI_StartScan()** and deleted when the application calls **SI_EndScan()**. The filename is unique for each scan.

The spooler property will automatically be set as disabled if there is not enough disk space to hold the image. Therefore, you should ensure there is enough disk space at least in the case of duplex scanning, since the spooler must be enabled for duplex. If it is disabled with duplex scanning, you will read incomplete image data.

Function Reference

The scanner is controlled through a set of functions exported from the DLL. The functions are callable from C or C++. All function names begin with the prefix “SI_”.

All API functions return a result code of type `SIResult` which is defined as:

```
typedef uint32 SIResult;
```

The result code normally reports the success or failure of the function. Possible result codes are defined in the ScannerAPI.h header file and begin with the prefix “`SIR_`”. In general, the code `SIR_SUCCESS` is returned when a function completes successfully.

All API functions and callback functions use the C calling convention (__cdecl). In case you use a different calling convention in your application’s compiler settings, you should specify the function pointers with the SICALLTYPE macro. For example, you would define SI_OpenInterface as:

```
typedef SIResult  (SICALLTYPE* SI_OpenInterfaceProc)(const char* pName);
```

Callback functions should use the SICALLBACK macro. See the **SI_Calibrate** or **SI_Clean** functions for more information.

SI_OpenInterface

The SI_OpenInterface function opens and initializes the scanner interface for use. It must be called before any other function.

```
SIResult SI_OpenInterface(
      const char* pName
);
```

Parameters

pName	[in] A pointer to an ASCII string identifying the scanner to open. This string is case-sensitive. The string is product-specific. For the exact string to use for a given scanner, check the addendum document for that scanner model.

Return Values

SIR_SUCCESS	The scanner was opened successfully.
SIR_ALREADY_OPEN	SI_OpenInterface has already been called and the scanner interface is already open.
SIR_UNKNOWN_MODEL_NAME	The string passed in with the pName parameter was not recognized as a supported model.
SIR_BAD_PARAMETER	pName was NULL.
SIR_SCANNER_NOT_READY	The scanner hardware could not be detected. The scanner is not connected.
SIR_INITIALIZATION_FAILURE	The scanner could not be opened and initialized.
SIR_DEVICE_COMMUNICATION_ERROR	There was a low-level error while trying to communicate with the scanner. The driver could not communicate with the scanner.

Remarks

SI_OpenInterface must be the first function that is called. It initializes the scanner and opens the API for use. The application should **call SI_CloseInterface** when the scanner is not longer needed.

See Also

SI_CloseInterface

SI_CloseInterface

This **SI_CloseInterface** function closes the scanner API and frees resource used by the scanner. It is the complementary function to **SI_OpenInterface**.

```
SIResult SI_CloseInterface();
```

Parameters

None.

Return Values

SIR_SUCCESS	The scanner interface was closed successfully.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.

Remarks

SI_CloseInterface must be called before the DLL is unloaded to free resources. After calling **SI_CloseInterface**, no other functions can be called (until **SI_OpenInterface** is called again).

See Also

SI_OpenInterface

SI_IsCalibrated

The SI_IsCalibrated function determines whether or not the scanner has been calibrated.

```
SIResult SI_IsCalibrated(
     SICalibrationState *pState
);
```

Parameters

pState	[out] On successful return, *pState is set to SI_TRUE if the scanner is calibrated, SI_FALSE if the scanner is not calibrated.

Return Values

SIR_SUCCESS	The calibration state was returned successfully. The value pointed to by pState has been set.
SIR_BAD_PARAMETER	The *pState* parameter was NULL.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.

Remarks

To verify that calibration has been done, **SI_IsCalibrated** checks for the existence of a calibration data file on disk. See **SI_Calibrate** for more info.

See Also

SI_Calibrate

SI_Calibrate

The SI_Calibrate function performs calibration on the scanner.

```
SIResult SI_Calibrate(
      uint32 target,
      ProgressCallbackPtr pProgressCallback
);
```

Parameters

target [in] A value that identifies the specific target that is inserted in the scanner. This value is model-specific. See Remarks.

Possible values for this parameter are:

SI_CT_DEFAULT	The scanner will perform the default calibration procedure. The calibration procedure will auto-detect the target depending on the model.

pProgressCallback [in] A pointer to a user function that will be called periodically during the calibration process to report the progress and to allow the application to cancel the calibration. This parameter can be set to NULL if no callback if needed. See Remarks for more details.

Return Values

SIR_SUCCESS	The scanner was calibrated successfully.
SIR_CALIB_WHITE_TARGET	The scanner calibrated successfully assuming the target was a white-only target. This will be returned when the calibration process is able to auto-detect the target type, and it detected that the target was white-only.
SIR_NOT_CALIBRATED	The scanner could not be calibrated.

SIR_BAD_PARAMETER	The value of the *target* parameter was not valid.
SIR_USER_CANCELLED	A callback function was specified in *pProgressCallback* and that callback returned SI_FALSE to cancel the calibration process.
SIR_FILESYSTEM_ACCESS_DENIED	The file system did not allow access to the calibration file.
SIR_SCANNER_BUSY	The scanner is busy and cannot perform calibration.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.
SIR_DEVICE_COMMUNICATION_ERROR	There was a low-level error while trying to communicate with the scanner. The driver could not communicate with the scanner.

Remarks

For scanner models that can use more than one possible calibration target—for example, black & white or white-only—the *target* parameter can be set by the application to specify which target is inserted. This parameter is therefore model-specific. Some models have only one possible target and so this parameter is ignored. Some models can accept more than one target but the calibration process can auto-detect which target is inserted. In such a case, you should pass SI_CT_DEFAULT as the *target* parameter.

If the *pProgressCallback* parameter is set to NULL, the **SI_Calibrate** function will not return until calibration is completed or an error occurred. The time to complete calibration may vary depending on the scanner model.

If the *pProgressCallback* parameter is not NULL, it must point to a function that will be called during the calibration process to report the percentage complete. The function must have the following prototype:

```
int32 SICALLBACK ProgressCallback(uint32 percentCompleted);
```

This function will be called with the *percentComplete* parameter indicating the progress of the calibration as a percentage (from 0 to 100). To allow the calibration to continue, you must return `SI_TRUE`. If `SI_FALSE` is returned, the calibration will be aborted and the **SI_Calibrate**

function will return `SIR_USER_CANCELLED`. (Remember that if you are using C++, the callback function must be static.)

Note that after calling **SI_Calibrate**, all properties will be reset to their default values.

Calibration data is stored in a file named `Calibration.dat`. The location of this file on disk depends on the operating system. On Windows systems, the file is stored in the folder indicated by the Windows API function **SHGetFolderPath()** using the folder ID `CSIDL_COMMON_APPDATA`. Therefore, the file will exist in a location common to all users on the computer. The file will normally be placed in a subfolder of that location based on the scanner model name. This folder will be created when SI_OpenInterface is called if it does not already exist, provided that the user has sufficient permission. The **SI_IsCalibrated** function will check for the existence of this file to determine if the scanner has been calibrated or not. The administrator of the system can set the permissions of the file to prevent writing by other users if those users should not be allowed to calibrate the scanner.

See Also

SI_IsCalibrated

SI_Clean

The SI_Clean function performs a back-and-forth feeding motion to clean the glass of the image sensor module when special cleaning paper is used.

```
SIResult SI_Clean(
     ProgressCallbackPtr pProgressCallback
);
```

Parameters

pProgressCallback [in] A pointer to a user function that will be called periodically during the cleaning process to report the progress and to allow the application to cancel the operation. This parameter can be set to

NULL if no callback if needed. See Remarks for more details.

Return Values

SIR_SUCCESS	The cleaning process completed successfully.
SIR_USER_CANCELLED	A callback function was specified in *pProgressCallback* and that callback returned SI_FALSE to cancel the cleaning process.
SIR_SCANNER_BUSY	The scanner is busy and cannot perform cleaning.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.
SIR_DEVICE_COMMUNICATION_ERROR	There was a low-level error while trying to communicate with the scanner. The driver could not communicate with the scanner.

Remarks

Before calling the **SI_Clean** function, you must check the paper sensor (see **SI_GetPaperStatus**) make sure the user has inserted the special cleaning paper. **SI_Clean** does not check for paper.

If the *pProgressCallback* parameter is set to NULL, the **SI_Clean** function will not return until cleaning is completed or an error occurred. The time to complete cleaning may vary depending on the scanner model.

If the *pProgressCallback* parameter is not NULL, it must point to a function that will be called during the cleaning process to report the percentage complete. The function must have the following prototype:

```
int32 SICALLBACK ProgressCallback(uint32 percentCompleted);
```

This function will be called with the *percentComplete* parameter indicating the progress of the cleaning process as a percentage (from 0 to 100). To allow the cleaning to continue, you must return `SI_TRUE`. If `SI_FALSE` is returned, the cleaning will be aborted and the **SI_Clean**

function will return `SIR_USER_CANCELLED`. (Remember that if you are using C++, the callback function must be static.)

SI_GetScannerStatus

The SI_GetScannerStatus returns information about the current status of the scanner.

```
SIResult SI_GetScannerStatus(
     SIScannerStatus *pScannerStatus
);
```

Parameters

pScannerStatus [out] A pointer to an SIScannerStatus variable which, upon successful return, is filled with a code indicating the scanner's current status. Possible values reported are:

SI_SS_OFFLINE	The scanner is not available.
SI_SS_ONLINE_READY	The scanner is online and ready to scan.
SI_SS_ONLINE_BUSY	The scanner is online but is busy and cannot scan. This can occur if the scanner is currently scanning in another process.

Return Values

SIR_SUCCESS	The scanner status was obtained successfully.
SIR_BAD_PARAMETER	The *pScannerStatus* pointer was NULL.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called. The status cannot be determined

Remarks

SI_GetScannerStatus can be called just prior to **SI_StartScan** to determine if the scanner is ready to scan. However, it is not strictly necessary to call **SI_GetScannerStatus** since **SI_StartScan** will return an appropriate error if the scanner is not ready for any reason. The best use of this function might be to immediately alert the user that the cable has been unplugged rather than wait until the next scan. In that case you could poll this function.

This function will not return SIR_DEVICE_COMMUNICATION_ERROR since if there is any problem communicating with the scanner, the status will be considered SI_SS_OFFLINE and the function will return SIR_SUCCESS.

See Also

SI_GetPaperStatus, SI_GetButtonStatus

SI_GetPaperStatus

The SI_GetPaperStatus indicates whether or not paper is inserted into the scanner.

```
SIResult SI_GetPaperStatus(
      SIPaperStatus *pPaperStatus
);
```

Parameters

pPaperStatus — [out] A pointer to an SIPaperStatus variable which, upon successful return, is filled with a code indicating whether or not paper is present. Possible values reported are:

SI_PS_PAPER_IN	Paper is inserted in the scanner.
SI_PS_PAPER_OUT	There is no paper in the scanner.

Return Values

SIR_SUCCESS	The scanner status was obtained successfully.
SIR_BAD_PARAMETER	The *pPaperStatus* pointer was NULL.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called. The paper status cannot be determined.
SIR_DEVICE_COMMUNICATION_ERROR	There was a low-level error while trying to communicate with the scanner. The driver could not communicate with the scanner.

Remarks

SI_GetPaperStatus can be called just prior to **SI_StartScan** to determine if there is paper in the scanner before starting a scan. **SI_StartScan** does not required that there be paper in the scanner, so it is up to the application to check for paper and only call **SI_StartScan** once paper is present.

SI_GetPaperStatus can be called at any time after **SI_OpenInterface** is called, even during scanning, between calls to **SI_ReadImageData**. Using it this way, one can detect when the end of page has been reached. However, that method can be inaccurate since the image is normally spooled to disk during scanning. The recommended method is to use the SIP_EOP_DETECT_ENABLED property.

See Also

SI_GetScannerStatus, SI_GetButtonStatus

SI_GetButtonStatus

The SI_GetButtonStatus indicates whether or not paper is inserted into the scanner.

```
SIResult SI_GetButtonStatus(
      uint32 buttonNumber,
      SIButtonStatus *pButtonStatus
);
```

Parameters

buttonNumber [in] A zero-based value indicating which button to check. The first button is 0, second button 1, and so forth. This value must be set even if there is only a single button available.

pButtonStatus [out] A pointer to an SIButtonStatus variable which, upon successful return, is filled with a code indicating whether or not the button is pressed. Possible values reported are:

SI_BS_UP	The button is not pressed.
SI_BS_DOWN	The button is pressed.

Return Values

SIR_SUCCESS	The scanner status was obtained successfully.
SIR_BAD_PARAMETER	The *pButtonStatus* pointer was NULL.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called. The button status cannot be determined
SIR_DEVICE_COMMUNICATION_ERROR	There was a low-level error while trying to communicate with the scanner. The driver could not communicate with the scanner.

Remarks

SI_GetButtonStatus can be called at any time after **SI_OpenInterface** is called, even during scanning, between calls to **SI_ReadImageData**.

See Also

SI_GetScannerStatus, SI_GetPaperStatus

SI_GetProperty

The SI_GetProperty function retrieves information about a specified scanner property. Examples of scanner properties are scan width and length, scan mode, and resolution.

```
SIResult SI_GetProperty(
      SIProperty* pProperty
);
```

Parameters

pProperty	[in-out] A pointer to an SIProperty structure. The propertyID member of this structure must be set to the ID of the property to retrieve.

Return Values

SIR_SUCCESS	The property information was obtained successfully.
SIR_BAD_PARAMETER	The *pProperty* pointer was NULL.
SIR_PROPERTY_UNSUPPORTED	The property ID set in the SIProperty structure was not one of the supported IDs for this scanner.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.

Remarks

Upon successful completion of the function, this structure pointed to by *pProperty* will be filled out with relevant information about that property, including the current value and valid values.

See the section on Properties for more information on getting and setting properties and a list of scanner properties that can be queried.

See Also

SI_SetProperty

SI_SetProperty

The SI_SetProperty function is used to set properties (scan parameters) used in scanning. Examples of scanner properties are scan width and length, scan mode, and resolution.

```
SIResult SI_SetProperty(
      SIProperty* pProperty
);
```

Parameters

pProperty	[in] A pointer to an SIProperty structure. Four fields of the property must be set: • *propertyID* • *containerType* • *itemType* • the *current* value (within the appropriate container)

Return Values

SIR_SUCCESS	The scanner status was obtained successfully.
SIR_BAD_PARAMETER	The *pProperty* parameter was NULL.
SIR_PROPERTY_UNSUPPORTED	The property ID set in the SIProperty structure was not one of the support IDs.
SIR_PROPERTY_INVALID_VALUE	One or more of the fields in the *pProperty* structure were invalid or out of range. Use SI_GetProperty to find valid values.

SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.

Remarks

Upon successful completion of the function, the property will be set to the new value. As a result, other properties may have changed. The application can use **SI_GetProperty** to check for any updated values.

See the section on Properties for more information on getting and setting properties and a list of scanner properties that can be set.

See Also

SI_GetProperty

SI_StartScan

The SI_StartScan function initiates a scan using the scan parameter set up in SI_SetProperty.

```
SIResult SI_StartScan();
```

Parameters

None.

Return Values

SIR_SUCCESS	The scan was initiated status was obtained successfully.

SIR_SCANNER_BUSY	The scanner is busy and cannot scan. Another process may currently be scanning.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.
SIR_DEVICE_COMMUNICATION_ERROR	There was a low-level error while trying to communicate with the scanner. The driver could not communicate with the scanner.

Remarks

The **SI_StartScan** function returns immediately after the scan was started. The application can then call **SI_ReadImageData** to begin retrieving the image data.

To end the scan that was started, the application must call **SI_EndScan**.

The API functions **SI_Feed** and **SI_FeedPaperOut** cannot be called between **SI_StartScan** and **SI_EndScan**. Attempting to call these functions at that time will result in a SIR_SCANNER_BUSY error.

See Also

SI_ReadImageData, SI_EndScan

SI_ReadImageData

The SI_StartScan function initiates a scan using the scan parameter set up in SI_SetProperty.

```
SIResult SI_ReadImageData(
      uint8  *buffer,
      uint32 numberOfLinesToRead,
      uint32 pageNumber,
      uint32 *numberOfLinesReturned
);
```

Parameters

buffer	[out]A pointer to a buffer to hold the image data. The size of the buffer required can be determined from the scanner property `SIP_LINE_WIDTH_IN_BYTES`, which gives the length in bytes of one line. Therefore the buffer size in bytes must be (*`numberOfLinesToRead`* * `SIP_LINE_WIDTH_IN_BYTES`).
numberOfLinesToRead	[in] The number of lines requested.
pageNumber	[in] For single-sided scanners, this must always be 0. For double-sided scanners, set this to 0 to read data from the front and 1 to read data from the back.
numberOfLinesReturned	A pointer to a uint32 variable that will be set to the number of lines actually returned. This may be less than the number requested.

Return Values

SIR_SUCCESS	The function returned successfully with zero or more lines in the buffer.
SIR_BAD_PARAMETER	One or more of the parameters were invalid.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.
SIR_ENDOFDATA	There is no more data for the specified page.
SIR_DEVICE_COMMUNICATION_ERROR	There was a low-level error while trying to communicate with the scanner. The driver could not communicate with the scanner.

Remarks

The value returned in *numberOfLinesReturned* may be zero. This does not mean that there is no more data. It may only mean that the application has caught up to the scanner and there is no new data available yet. When there is no more image data for the page, SIR_ENDOFDATA will be

returned. If the property SIP_EOP_DETECT_ENABLED is supported and is enabled, SIR_ENDOFDATA will be return when the end of page is detected. In this case, *numberOfLinesReturned* will be zero. Further calls to **SI_ReadImageData** will return a result of SIR_ENDOFDATA with *numberOfLinesReturned* set to zero.

If the property SIP_EOP_DETECT_ENABLED is supported and is *not* enabled, then the scan length is fixed. That is, **SI_ReadImageData** will not return SIR_ENDOFDATA until the number of lines specified in the SIP_SCAN_LENGTH_IN_LINES property is returned. Therefore, in this case, the application does not need to check for SIR_ENDOFDATA.

For duplex scanners, the front and back pages can be read in any order. It's not necessary that you read all lines of one page before you read lines from the other page. For example, you can read 10 lines from page 0, then 10 lines from page 1, then 10 more lines from page 0 again. However, lines can only be read once and are read sequentially—you cannot move the "file pointer" back to re-read lines. In order to read both the front and back pages correctly, the spooler must be enabled (See the SIP_SPOOLER_ENABLED), which it is by default. So you don't need to specifically enable it.

See Also

SI_StartScan, SI_EndScan

SI_EndScan

The SI_EndScan function terminates a scan that was started by calling SI_StartScan.

```
SIResult SI_EndScan();
```

Parameters

None.

Return Values

SIR_SUCCESS	The scan was ended successfully.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.
SIR_SCANNER_BUSY	The scanner is busy and cannot feed. Another process may currently be scanning.
SIR_DEVICE_COMMUNICATION_ERROR	There was a low-level error while trying to communicate with the scanner. The driver could not communicate with the scanner.

Remarks

The **SI_EndScan** function may be called anytime after **SI_StartScan**; it is not necessary to read all the image data. When called, the scan is immediately stopped. The paper feeding stops. The paper is not fed out of the scanner. To feed the paper out, call the **SI_FeedPaperOut** function. After calling **SI_EndScan,** the **SI_ReadImageData** function may no longer be called (until the next scan).

Calling **SI_EndScan** while a scan is not is progress does not result in an error.

See Also

SI_StartScan, SI_ReadImageData

SI_Feed

The SI_Feed function feeds the paper a specified distance without performing a scan.

```
SIResult SI_Feed();
```

Parameters

None.

Return Values

SIR_SUCCESS	The function performed the feed successfully.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called. The paper status cannot be determined.
SIR_SCANNER_BUSY	The scanner is busy and cannot feed. Another process may currently be scanning.
SIR_DEVICE_COMMUNICATION_ERROR	There was a low-level error while trying to communicate with the scanner. The driver could not communicate with the scanner.

Remarks

The distance and direction of the feed are determined by three properties which must be set before calling **SI_Feed**.

SIP_YRESOLUTION	The Y resolution is set so that there is some context to the number of lines specified in SIP_YOFFSET. That is, feeding 100 lines at a resolution of 300 DPI means to feed the paper 1/3 of an inch.
SIP_YOFFEST	This property specifies the number of lines to feed the paper. The lines are in terms of resolution specified in the SIP_YRESOLUTION property.

SIP_FEED_DIRECTION	This property specifies the direction the paper will be moved. This property may or may not exist for a given model scanner. If it does not, then the paper will always feed forward.

The **SI_Feed** function does not return until the paper has been fed the specified distance (or an error has occurred).

SI_Feed always feeds the length specified regardless of the state of the paper sensor. To feed the paper until the paper sensor is clear, use **SI_FeedPaperOut**.

SI_Feed cannot be called between **SI_StartScan** and **SI_EndScan**. Attempting to call it at that time will result in a SIR_SCANNER_BUSY error.

See Also

SI_FeedPaperOut

SI_FeedPaperOut

The SI_FeedPaperOut function feeds the paper until the paper sensor is clear.

```
SIResult SI_FeedPaperOut();
```

Parameters

None.

Return Values

SIR_SUCCESS	The function performed the feed successfully.

SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called. The paper status cannot be determined.
SIR_SCANNER_BUSY	The scanner is busy and cannot feed. Another process may currently be scanning.
SIR_DEVICE_COMMUNICATION_ERROR	There was a low-level error while trying to communicate with the scanner. The driver could not communicate with the scanner.

Remarks

The direction of the feed is determined by one property which must be set before calling **SI_FeedPaperOut**.

SIP_FEED_DIRECTION	This property specifies the direction the paper will be moved. This property may or may not exist for a given model scanner. If it does not, then the paper will always feed forward.

The **SI_FeedPaperOut** function will not stop feeding exactly when the paper sensor is cleared. Instead it will feed a bit further to make sure the paper is clear of the scanner mechanism**.** This distance depends on the particular model since the distance required to clear a page depends on the hardware.

If there is no paper detected when **SI_FeedPaperOut** is first called, **SI_FeedPaperOut** will still feed the paper some minimum distance to ensure the paper is clear from the scanner mechanism.

The **SI_FeedpaperOut** function does not return until the paper has been fed out or until the maximum feed length has been reached (or an error has occurred). The maximum feed length is defined as the maximum valid value of the property SIP_YOFFSET.

SI_FeedPaperOut cannot be called between **SI_StartScan** and **SI_EndScan**. Attempting to call it at that time will result in a SIR_SCANNER_BUSY error.

See Also

SI_Feed

SI_Diagnostic

The SI_Diagnostic function performs diagnostic tests on the scanner hardware.

```
SIResult SI_Diagnostic(
      SIDiagTest  test,
      void*       pDiagInfo
);
```

Parameters

test	[in] An ID value that specifies which test to perform. See the Remarks section for a list of possible values.
pDiagInfo	[in] A pointer to a diagnostic info structure. The structure passed in depends which test is specified in the *test* parameter. This structure may contain both input and output values for the diagnostic test. See the Remarks section for more info.

Return Values

SIR_SUCCESS	The test result was returned successfully.
SIR_BAD_PARAMETER	The test specified was unknown or the pDiagInfo parameter was NULL.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.

SIR_DEVICE_COMMUNICATION_ERROR	The test could not be performed because there was a low-level error while trying to communicate with the scanner.

Remarks

The second parameter *pDiagInfo* is a pointer to structure; the particular structure passed in depends on the test being performed. The following table lists the valid tests and the structures which must be passed in for each test. The structures are defined in the ScannerAPI.h header file. The comments also describe any structure members which must be set up before calling the function, as well as result values returned in structure members.

test	*pDialogInfo*	Comment
`SI_DIAG_HW_PRESENCE`	`SIDiagHwPresenceInfo`	**Output**: the result of the test is returned in the `result` member variable. Possible results are: • `SI_DIAG_PASS` – the scanner is connected. • `SI_DIAG_FAIL` – the scanner is not connected.
`SI_DIAG_READ_PERFORMANCE`	`SIDiagReadPerfInfo`	**Input**: The caller must set the `readLengthInKB` member variable to the data length in KB to read. Valid values are between 1 and 100 KB. **Output**: The total time in microseconds to read the data from the scanner is returned in the `timeInMicroSec` member variable.

The following is an example of calling SI_Diagnostic() to do a Read Performance test.

```
SIDiagReadPerfInfo      diagReadPerfInfo;

// Specify a 20 KB data length to read.
diagReadPerfInfo.readLengthInKB = 20;

// Do the test.
SI_Diagnostic(SI_DIAG_READ_PERFORMANCE, &diagReadPerfInfo)

// Print the result.
printf("The time to read %d KB was %d us.",
            diagReadPerfInfo.readLengthInKB,
            diagReadPerfInfo.timeInMicroSec);
```

SI_GetEvent

The SI_GetEvent function will interpret a USB interrupt code and translate it into a bit-flag indicating the event or events that occurred. This function is normally used only by a driver that handles USB interrupt events, such as a Windows STI driver.

```
SIResult SI_GetEvent(
      SIEventInfo* pEventInfo
);
```

Parameters

pEventInfo	[in-out] A pointer to an SIEventInfo structure. The *eventDataSize* and *eventData* structure fields must be filled out before passing the structure. See Remarks for more detail.

Return Values

SIR_SUCCESS	The event code was recognized and successfully converted to test. The *eventName* field of the *pEventInfo* structure was filled with the event name.
SIR_BAD_PARAMETER	The *pEventInfo* parameter was NULL.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface has not been called.

Remarks

When an event occurs in the scanner, such as a button press or paper sensor trigger, the scanner will send a USB interrupt to the host computer. Included in the interrupt message may be one or more bytes that indicate which event occurred. This information is not standard; it can be unique

to the scanner. Thus if you are writing a driver to handle this interrupt, you may not know which scanner event occurred.

The **SI_GetEvent** function will translate the interrupt data into one or more bit flags indicating which event occurred. To call this function, you must pass a pointer to an SIEventInfo structure. This structure is defined in ScannerAPI.h as:

```
#define MAX_EVENT_DATA_SIZE  20

typedef struct
{
       /// [in] The size in bytes of the data in the eventData array.
       uint8   eventDataSize;

       /// [in] The data bytes return by the scanner in the USB interrupt phase.
       uint8   eventData[MAX_EVENT_DATA_SIZE];

       /// [out] Bit flags that indicate which event(s) occurred.
       uint32 eventFlags;

} SIEventInfo;
```

The caller must copy the data from the USB interrupt into the *eventData* array and also set the *eventDataSize* field to the number of bytes in that array. If SI_GetEvent returns successfully, the eventFlags field will have bits flags set indicating which event or events occurred. Possible bitflags set are:

SIEVT_PAPER_IN	The paper sensor was triggered. This indicates that the paper was inserted.
SIEVT_BUTTON_DOWN	The first scan button was pressed.
SIEVT_BUTTON2_DOWN	The second scan button was pressed.
SIEVT_BUTTON3_DOWN	The third scan button was pressed.

If the event data was not recognized, then the *eventFlags* field will be returned as 0. Some scanners do not support USB interrupts. In this case, the *eventFlags* field will be returned as 0.

See Also

SI_Reset

The SI_Reset function resets the scanner to a known state.

```
SIResult SI_Reset();
```

Parameters

None.

Return Values

SIR_SUCCESS	The scanner was reset successfully.
SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.
SIR_DEVICE_COMMUNICATION_ERROR	The scanner could not be reset because there was a low-level error while trying to communicate with the scanner.

Remarks

SI_Reset should be used with caution. It will reset the scanner hardware regardless of whether a scan is in progress, either in the current process or a different process. Resetting the hardware will abort any scan or feed in progress.

In addition, all scan properties are set to their default values.

SI_GetLastError

The SI_GetLastError function returns the result of the most recently called API function.

```
SIResult SI_GetLastError();
```

Parameters

None.

Return Values

The result that was returned by the most recent API function that was called.

Remarks

You typically do not need to use **SI_GetLastError** since the result codes are returned from the functions themselves. **SI_GetLastError** is included for convenience and may be useful depending on how your application is structured.

The interface does not need to be open to call **SI_GetLastError**.

See Also

SI_GetLastErrorText

SI_GetLastErrorText

The SI_GetLastError function returns the result of the most recently called API function.

```
SIResult SI_GetLastErrorText(char **ppErrorText);
```

Parameters

ppErrorText	[out] A pointer to a char pointer that will be set to point to a null-terminated text string that describes the last API result

that occurred. Do not attempt to free the memory pointed to by this pointer.

Return Values

SIR_SUCCESS	This pointer to the text string was returned successfully.
SIR_BAD_PARAMETER	The *ppErrorText* parameter was NULL.

Remarks

You typically do not need to use **SI_GetLastErrorText** since the result codes are returned from the functions themselves. **SI_GetLastErrorText** is included for convenience and may be useful for displaying error messages.

The interface does not need to be open to call **SI_GetLastErrorText**.

See Also

SI_GetLastError

Properties

Properties are a flexible mechanism for setting and retrieving scan parameters. There are properties for scan width, scan length, resolution, brightness, and other scanner parameters that can be queried and controlled. Properties are analogous to capabilities in the TWAIN protocol or to item properties in the Windows Image Acquisition protocol. However, DCT properties are somewhat simpler since a smaller subset of features needs to be supported.

Functions

You can get or set properties using two functions of the API:

```
SIResult SI_GetProperty(SIProperty* pProperty);
SIResult SI_SetProperty(SIProperty* pProperty);
```

SIProperty is a structure defined as:

```
typedef struct
{
	uint16		propertyID;		///< a property ID constant.
	uint16		containerType;		///< a container type constant.
	uint16		itemType;		///< the data type the container holds.
	uint16		access;			///< SIACC_READWRITE or SIACC_READONLY.

	// containers
	union
	{
		SISingle		single
		SIRange			range;
		SIList			list;
		SIArray			array;
	};
} SIProperty;
```

The *propertyID* field holds a property ID constant that specifies the property to set or get. Examples of property IDs are SIP_XRESOLUTION and SIP_SCAN_WIDTH_IN_PIXELS. All property ID constants begin with the prefix “SIP_”.

The *containerType* field is used to specify which type of container is used to store the property values. The anonymous union contains structures for all possible container types. The

containerType field specifies which of these is being used. This is set by the DLL on return from **SI_GetProperty**. It must also be set by the application before calling **SI_SetProperty.** Valid values for *containerType* are:

SICON_SINGLE	The property is a single value. A property with this container type will normally be read-only, since there is only one possible value. The exception is when the itemType is SI_BOOL, in which case the valid range of values is understood to be either SI_TRUE or SI_FALSE. When the itemType is SI_BOOL, the property may be either read-only or read-write.
SICON_RANGE	The property can be selected from a range of possible values. This container includes a minimum and maximum value allowed and also an increment.
SICON_LIST	This property can be selected from a list of specific possible values. The list of possible values is defined in the container.
SICON_ARRAY	This property is an array of values.

The *itemType* field describes the data type of the value. This is set by the DLL on return from **SI_GetProperty**. Valid values for *itemType* are:

SI_INT32	The value is a 4-byte signed integer.
SI_FLOAT32	The value is a 4-byte floating point value.
SI_STR	The value is a pointer to a null-terminated ASCII string.
SI_BOOL	The value is either SI_TRUE or SI_FALSE.

The *access* field specifies whether the application is allowed to change the current value of the property. This is set by the DLL on return from **SI_GetProperty**. Possible values are:

SIACC_READONLY	The current value cannot be set.
SIACC_READWRITE	The current value can be set.

The *access* field is primarily of use when the container type is SICON_SINGLE and the item type is SI_BOOL. In the current implementation of the DLL, access on non-bool single containers is always SIACC_READONLY and access of all other containers is SIACC_READWRITE. (In these cases, the *access* field is included for possible future use.)

Values and Containers

SIValue

All values are stored in an *SIValue* structure (which is in turn stored in a container). Using a union, *SIValue* can store any basic type used by properties.

```
typedef struct
{
      union
      {
            int32       iVal;       ///< integer value
            float32     fVal;       ///< float value
            char*       strVal;     ///< string value
            int32       bVal;       ///< Boolean. SI_TRUE or SI_FALSE

            int32*      piVal;      ///< pointer to integer
            int16*      psVal;      ///< pointer to short
            int8*       pcVal;      ///< pointer to character
            char**      pstrVal;    ///< pointer to string
      };
} SIValue;
```

The *iVal* and *fVal* members store integer and floating point values respectively. Typically, the *int32* type is used for all integer values for consistency.

The *strVal* is a pointer to a NULL-terminated ASCII string. Thus the string is not stored within the property container itself. The pointer normally points to a statically allocated string in the DLL memory.

The members *piVal*, *psVal*, *pcVal*, *and* `pstrVal` are used only in list and array containers, described below. They are not used in SISingle and SIRange containers. They point to the list or array itself. Using these fields helps to avoid the need for type casts.

SISingle

An *SISingle* container uses the following structure.

```
typedef struct
{
      SIValue     current;
      SIValue     def;
} SISingle;
```

This container holds a single value, the *current* field. Since there is no range of possible choices for the value, it is a read-only value and cannot be changed (unless the *itemType* is SI_BOOL). The access field will be SIACC_READONLY. The *def* field holds a default value which will always be the same as the current value except when the *itemType* is SI_BOOL.

When the *itemType* is SI_BOOL, the application may set the current value. In this case, the possible choices are understood to be SI_TRUE or SI_FALSE. So for SI_BOOL types, the access field may be either SIACC_READONLY or SIACC_READWRITE. The *def* value might not be the same as the *current* value if the *current* value was changed.

SIRange

An *SIRange* container uses the following structure:

```
typedef struct
{
      SIValue     minimum;
      SIValue     maximum;
      SIValue     stepSize;
      SIValue     def;
      SIValue     current;
} SIRange;
```

This container can hold any value within a range defined by the *minimum* and *maximum* value and divided up into equal-sized steps. The *stepSize* field specifies the size of the steps. So for example, if the scanner's width was defined and a range with minimum 0, maximum 1000 and step size 10, then you could select widths of 0, 10, 20, 30,…, 990, 1000.

The *current* field holds the current value and the *def* field holds the default value.

List

An *SIList* container uses the following structure:

```
typedef struct
{
      int32 numItems;
      SIValue     current;
      SIValue     def;
      SIValue     items;
} SIList;
```

An SIList container defines a list of possible values from which the current value can be chosen. Unlike an SIRange container, the items in the list are not necessarily spaced equally.

The *items* field holds a pointer to the item list. The type of pointer depends on the item type specified in the *itemType* field of the property. For example, if the items type is SI_INT32, then the pointer is stored in the piVal member of the SIValue structure. You could access the third item in the list with the expression:

```
int32 thirdValue = prop.list.items.piVal[2];
```

The *current* field stores the current item itself, not an index into the list of items (as TWAIN does, for example). To set the current value to be the first item in the list, you would use the expression:

```
prop.list.current.iVal = prop.list.items.piVal[0];
```

In the case of a string type, the current value must match one of the string pointers in the items list. It cannot be a different pointer to a string that happens match one in the list; that is, one that would compare equally using the strcmp() library function. Therefore, if the item type is a string, to set the current value to be the first item in the list you would use the expression:

```
prop.list.current.strVal = prop.list.items.pstrVal[0];
```

The *numItems* field specifies the number of items in the list.

The *def* field holds the default value.

SIArray

An *SIArray* container uses the following structure:

```
typedef struct
{
      int32 numItems;
      SIValue     items;
} SIArray;
```

An SIArray container defines an array of items. The *items* field holds a pointer to the array values. The type of pointer depends on the item type specified in the *itemType* field of the property. For example, if the items type is SI_INT32, then the pointer is stored in the piVal member of the SIValue structure. You could access the item at index 100 using the expression:

```
int32 item100 = prop.array.items.piVal[100];
```

The *numItems* field specifies the number of items in the array. There is no default value for arrays.

Retrieving Property Values

Retrieving a scanner property value can best be shown with a short example. To retrieve a property, you need to set the propertyID member of the property to the ID you want to retrieve. Property ID identifiers all begin with the prefix “SIP_”. Then call the **SI_GetProperty** function.

```
SIProperty  prop;
SIResult    result;

prop.propertyID = SIP_XRESOLUTION;        // initialize the ID

result = SI_GetProperty(&prop);           // Get the property

assert(result == SIR_SUCCESS);            // make sure it succeeded
```

Upon return, the `prop` structure will be filled in with information for that property. You can then examine the *containerType* field. This tells you which container structure the property info is stored in. The *itemType* field will tell you which data type the value is, so you can access it with the correct member of the Value structure.

```
// The X resolution will always be an integer type.
assert(prop.itemType == SI_INT32);

// Check the container type and print out the current resolution
switch(prop.containerType)
{
   case SICON_SINGLE:
      printf("The current resolution is %d\n", prop.single.current.iVal);
      break;
   case SICON_LIST:
      printf("The current resolution is %d\n", prop.list.current.iVal);
      break;
   case SICON_RANGE:
      printf("The current resolution is %d\n", prop.range.current.iVal);
      break;
   default:
      // You should never get here.
      // An array container would not be used.
      break;
};
```

In the above example, since the application examines the *containerType* field first, it can deal with the property no matter which type of container was returned by **SI_GetProperty**. However, if you're developing an application for a specific model of scanner, you normally know the container type and item type. Therefore you can avoid checking these and instead just write:

```
// Assume the X resolution will always be a list container and integer
// type.
assert((prop.containerType == SICON_LIST) && (prop.itemType == SI_INT32));

printf("The current resolution is %d\n", prop.list.current.iVal);
```

It's up to the application developer. If you intend to develop an application to handle more than one scanner model, and the models use different containers and item types, then you may want to write the application code in a more abstract and flexible way. But in most cases, you can just assume the container and item type once you know them for a particular property.

Setting Property Values

To set a property, you must specify the following four things in the property structure:

- The property ID in the propertyID field.
- The container type.
- The item type
- The new current value.

All other fields will be ignored. This means, for example, that you cannot set the minimum or maximum values of range types. Nor can you set the default value. All these are fixed by the DLL.

The following example sets the X offset property (the left margin).

```
SIProperty  prop;
SIResult    result;

// Initialize property fields
prop.propertyID = SIP_XOFFSET;
prop.containerType = SICON_RANGE;
prop.itemType = SI_INT32;
prop.range.current.iVal = 0;

// Set the property's current value
result = SI_SetProperty(&prop);
assert(result == SIR_SUCCESS);
```

Instead of setting the container type and item type yourself, you could also have the DLL set these by calling **SI_GetProperty()**.

```
SIProperty  prop;
SIResult    result;

// Initialize property fields
prop.propertyID = SIP_XOFFSET;

// Get the property's info
result = SI_GetProperty(&prop);
assert(result == SIR_SUCCESS);

// Set the property's current value. Assume the previous call to
// SI_GetProperty() has already set the container type to range and
// the item type to integer.
prop.range.current.iVal = 0;
result = SI_SetProperty(&prop);
assert(result == SIR_SUCCESS);
```

When you specify the item type to **SI_SetProperty**, the types must match the documented types for that property. If they do not, **SI_SetProperty** will return an SIR_PROPERTY_INVALID_VALUE error. The container type must also match, with the exception that you can also use the SICON_SINGLE container to set the current value for properties that are SICON_RANGE or SICON_LIST types. So, for example, even though the

SIR_XOFFSET property shown below returns a SICON_RANGE container from **SI_GetProperty**, you can use a SICON_SINGLE container to set its current value.

```
SIProperty  prop;
SIResult    result;

// Initialize property fields
prop.propertyID = SIP_XOFFSET;
prop.containerType = SICON_SINGLE;
prop.itemType = SI_INT32;
prop.single.current.iVal = 0;

// Set the property's current value
result = SI_SetProperty(&prop);
assert(result == SIR_SUCCESS);
```

If you are developing for more than one scanner model, and you want to develop the application in a flexible way, you can call **SI_GetProperty** to discover the container and item types and then set the current value appropriately. The following example might be the case if you're writing code that handles two different scanners. One scanner fixes the X offset at 0 and uses the Single container. The other allows a range of values and uses the range container.

```
SIProperty  prop;
SIResult    result;
int32 newOffset = 100;

// Get the property's info
prop.propertyID = SIP_XOFFSET;
result = SI_GetProperty(&prop);
assert(result == SIR_SUCCESS);

// Set the property's current value.
switch(prop.containerType)
{
   case SICON_SINGLE:

      // Only one value of x Offset is allowed. So we can't set it.
      printf("The X Offset %d can't be changed.\n", prop.single.current.iVal);
      break;

   case SICON_RANGE:

      // Check that the value is valid.
      if(   (newOffset >= prop.range.minimum.iVal) &&
            (newOffset <= prop.range.maximum.iVal) &&
            ((newOffset % prop.range.stepSize.iVal) == 0))
      {
            prop.range.current.iVal = newOffset;
            result = SI_SetProperty(&prop);
            assert(result == SIR_SUCCESS);
            printf("The X Offset value was changed to %d.\n", newOffset);
      }
      else
      {
            printf("The X Offset value is not valid for this property.\n");
      }
      break;

};
```

Again, as in retrieving properties, it's usually not necessary to go to these lengths if you're developing for only one scanner model and you know the container and items types.

Note that when one property is changed by the application, it may result in a change of the current values or valid ranges of other properties. For example, a scanner may have an allowable scan width of 2592 pixels at 300 DPI resolution. If the application changes the resolution to 100 DPI, the allowable scan width may then be reduced to 864 pixels. In addition, if the current scan width value was set higher than 864 pixels, it will be reduced to 864 to be within the valid width. Therefore, after setting scan properties, the application should read back the relevant properties that may have changed to confirm their values.

For the purpose of these adjustments, properties have the following priority:

1. SIP_SCAN_MODE
2. SIP_BITS_PER_CHANNEL
3. SIP_XRESOLUTION, SIP_YRESOLUTION
4. SIP_XOFFSET, SIP_YOFFSET
5. SIP_SCAN_WIDTH_IN_PIXELS, SIP_SCAN_LENGTH_IN_LINES

As another example, suppose a scan has a maximum scan width of 2592 pixels at 300 DPI and the current value is set to the maximum 2592. The X offset could then be set to 100. Then the maximum and current scan width would be reduced to 2492. The current or maximum value for scan width could not be set to 2592 and force the offset back to 0 because the X offset property has higher priority.

Property Reference

Not all scanner models will have all properties. Some properties may not be appropriate for the given scanner's capabilities. To programmatically find out if a scanner supports a given property, you can call SI_GetProperty as shown:

```
SIProperty  prop;
SIResult    result;

prop.propertyID = SIP_XRESOLUTION;        // initialize the ID

result = SI_GetProperty(&prop);           // Get the property

if(result == SIR_SUCCESS)
{
      printf("The property is supported.\n");
}
else if(result == SIR_PROPERTY_UNSUPPORTED)
{
      printf("The property is not supported.\n");
}
else
{
      // some other error occurred.
}
```

Scanners typically have at least the following properties which are needed for basic scanning.

```
SIP_OPTICAL_RESOLUTION
SIP_OPTICAL_WIDTH_IN_PIXELS
SIP_XRESOLUTION
SIP_YRESOLUTION
SIP_BITS_PER_CHANNEL
SIP_BITS_PER_PIXEL
SIP_SCAN_MODE
SIP_YOFFSET
SIP_XOFFSET
SIP_SCAN_WIDTH_IN_PIXELS
SIP_SCAN_LENGTH_IN_LINES
SIP_LINE_WIDTH_IN_BYTES
```

You do not need to set all properties. You can ignore properties that you don't care about or for which the default value is fine. Normally you'll want to set the basic properties like `SIP_SCAN_MODE` or `SIP_XRESOLUTION`.

SIP_BITS_PER_CHANNEL

Description

The bit depth per color channel. For example, if a scan mode supports 24-bit color, then this property value would be set to 8 since each of the red, green, and blue channels are 8-bits deep.

Most often, this is a read-only value using a SICON_SINGLE container. But it may possibly be writable and use a SICON_LIST container for a scanner that provides a choice of bit depth, for example 4-bit or 8-bit grayscale.

Item Types

SI_INT32

Container Types

SICON_SINGLE
SICON_LIST

SIP_BITS_PER_PIXEL

Description

The number of bits per pixel. This includes all channels for scan modes with more than one color channel.

Most often, this is a read-only value.

Item Types

SI_INT32

Container Types

SICON_SINGLE

SIP_BRIGHTNESS

Description

Controls the brightness level of the image. For B&W scan mode, this is equivalent to lowering the threshold.

Item Types

SI_INT32

Container Types

SICON_RANGE

SIP_CHANNEL_ORDER

Description

Specifies the order of color channels. Since most scanners provide data in RGB order, this can be useful when saving to Windows Bitmap format which require BGR order. Channel order applies only when SIP_PLANARCHUNKY property is SI_PC_CHUNKY.

If the container is SICON_SINGLE, then the order cannot be changed.

Item Types

SI_INT32

Container Types

SICON_SINGLE
SICON_LIST

Values Allowed

SI_CO_RGB	color channels are ordered red-green-blue
SI_CO_BGR	color channels are ordered blue-green-red

SIP_CONTRAST

Description

The contrast of the image.

The allowable range is centered on zero, with zero being the nominal value. Positive values provide higher contrast and negative values lower contrast.

Item Types

SI_INT32

Container Types

SICON_RANGE

SIP_DESCREEN_ENABLED

Description

Specifies whether or not descreen processing is enabled. The descreen process will reduce the moiré effects that result from scanning a screened target such as magazine or newspaper image. When set to SI_TRUE, the descreen function is enabled. The default value is SI_FALSE.

Item Types

SI_BOOL

Container Types

SICON_SINGLE

Values Allowed

SI_TRUE
SI_FALSE

SIP_DROPOUT_COLOR

Description

Specifies the dropout color for scanning. The dropout color is the color that is not scanned. This color will show up as white. It can be used in scanning forms for better OCR processing.

This setting applies only to grayscale and B&W scan modes. It is ignored for color scan mode.

Item Types

SI_INT32

Container Types

SICON_SINGLE
SICON_LIST

Values Allowed

SI_DO_NONE (default)
SI_DO_RED
SI_DO_GREEN
SI_DO_BLUE

SIP_DUPLEX_ENABLED

Description

Specifies whether or not duplex (double-sided) scanning is enabled. If this property does not exist, then duplex scanning is not supported on the scanner. Single-sided scanners will not have this property.

When set to SI_TRUE, the scanner will scan both sides of a page. This normally results in a reduced scan speed compared to single-sided scanning.

When this property is set to SI_TRUE, the SI_SPOOLER_ENABLED property will be changed to SI_TRUE if it was previously set to SI_FALSE. This is because the spooler is required to be enabled for duplex scanning.

See the **SI_ReadImageData()** function for information on reading image data from a duplex scan.

Item Types

SI_BOOL

Container Types

SICON_SINGLE

Values Allowed

SI_TRUE
SI_FALSE

SIP_EOP_DETECT_ENABLED

Description

Enables end-of-page detection.

When set to SI_TRUE, the paper sensor will be monitored and the scan will stop when the paper is clear. Actually, the scan will not stop as soon as the paper has cleared the paper sensor, but rather as soon as it clears the image sensor which will be some time later depending on the distance between paper sensor and image sensor and whether or not duplex scanning is enabled.

When end-of-page detection is enabled, the application reads data normally, using **SI_ReadImageData()**. The end of the data will be reached when SIR_ENDOFDATA is returned from the function. See the **SI_ReadImageData()** function for more information.

When the end of the page is reached, the DLL will not feed the paper completely out of the scanner. The application must call the **SI_FeedPaperOut()** function to do this.

Even when end-of-page detect is enabled, a scanned page will never be longer than the length specified in the SIP_SCAN_LENGTH_IN_LINES property. Thus, the value of SIP_SCAN_LENGTH_IN_LINES should represent the maximum length you want to scan.

Item Types

SI_BOOL

Container Types

SICON_SINGLE

Values Allowed

SI_TRUE
SI_FALSE

SIP_EOP_DETECT_OFFSET

Description

Adjusts the bottom point of the page when end-of-page detection is enabled.

If you find that the bottoms of pages are getting clipped too much by the end-of-page detection feature, you can adjust the bottom point using this property. Positive values make the page longer and negative values make the page shorter. Units are in hundredths of an inch. Maximum offset is one inch in either direction. The default is zero.

Regardless of the offset specified in this property, a scanned page is never longer than the length specified in the SIP_SCAN_LENGTH_IN_LINES property.

Item Types

SI_INT32

Container Types

SICON_RANGE

SIP_FEED_DIRECTION

Description

Sets the feed direction, either forward or reverse, for scanning and feeding.

When feeding the paper, this property should be used in conjunction with SIP_YRESOLUTION and SIP_YOFFSET. These three properties define the feeding characteristics. Then **SI_Feed()** can be called to perform the feed. Reverse feeding is useful for rejecting the paper when as scan is canceled by the user. In that case, the feed direction can be set to reverse and **SI_FeedPaperOut()** can be called. The paper will be fed in reverse until is clears the paper sensor.

Whether or not the scanner allows the reverse direction for both feeding and scanning depends on the particular model. Some models may allow reverse feeding but not reverse scanning.

If this property is not supported, the scanner can only feed forward.

Item Types

SI_INT32

Container Types

SICON_LIST

Values Allowed

SI_FORWARD
SI_REVERSE

SIP_FEED_RATE

Description

Allows adjustment of the feeding speed in inches per second. By default, feeds will run at the fastest speed possible for a given mode and resolution. Note that feed rate is not the same as scan rate (see the SIP_SCAN_RATE property).

This property is optional and need not be changed. But in some cases, you may want to purposely slow the feeding speed if the default is too fast for your particular application.

When you retrieve this property, the current value returned is the speed that paper will move in *inches per second*. Unlike the SIP_SCAN_RATE property, the feed rate does not depend on the settings of any other properties such as scan mode, resolution, or duplex. Since this property uses a range container, you'll also get a minimum and maximum value that this property can be set to. Setting the current value lower than the maximum will result in slower feed rates.

Adjusting this property will change the *feeding* portion of a scan but not the scan speed itself. Thus if you scan a page with a top offset, the page will first be fed a distance equal to the top offset at the feed rate set by this property. Then the scanning portion proceeds at the speed set by the SIP_SCAN_RATE property.

The SIP_FEED_RATE property may or may not be supported depending on the scanner model.

Item Types

SI_FLOAT32

Container Types

SICON_RANGE

SIP_GAMMA

Description

Sets the gamma value. The gamma value is used to create a lookup table that represents a gamma curve.

Note that when any custom lookup table has been set with the properties SIP_LUT_RED, SIP_LUT_GREEN, SIP_LUT_BLUE, or SIP_LUT_GRAY, the gamma value set in this property is ignored.

Item Types

SI_FLOAT32

Container Types

SICON_RANGE

SIP_HIGHLIGHT

Description

Sets the highlight level. The highlight level is the maximum white level. So for an 8-bit grayscale image, for example, this would normally be set to 255 by default. If this value is reduced to a lower value, then image levels that were originally darker will appear brighter.

Note the interaction between SIP_HIGHLIGHT and SIP_SHADOW when setting the current value. These two properties together define the range from dark to light. SIP_HIGHLIGHT can never be equal to or less than the value of SIP_SHADOW. So changing the current value of SIP_ HIGHLIGHT may cause the current SIP_ SHADOW to be automatically adjusted to be one less than SIP_ HIGHLIGHT if it was originally greater than or equal to it. You can check the current value of SIP_ SHADOW to confirm this. Likewise, changing the current value of SIP_ SHADOW may affect the current value of SIP_ HIGHLIGHT.

When any custom lookup table has been set with the properties SIP_LUT_RED, SIP_LUT_GREEN, SIP_LUT_BLUE, or SIP_LUT_GRAY, the highlight value set in this property is ignored.

Item Types

SI_INT32

Container Types

SICON_RANGE

SIP_LED_INDICATOR1

SIP_LED_INDICATOR2

Description

Turns the LED indicators on or off. These LEDs are typically used to indicate some status of the scanner, such as power on, ready so scan, or busy scanning. The number, location, and color of the LEDs depend on the particular scanner model. Many models do not have LEDs.

You can call **SI_GetProperty()** to find out the current state of the LEDs. If the scanner has no LEDs, then **SI_GetProperty()** will return SIR_PROPERTY_UNSUPPORTED.

Item Types

SI_BOOL

Container Types

SICON_SINGLE

SIP_LINE_WIDTH_IN_BYTES

Description

Reports the line width, in terms of bytes, that will be returned by **SI_ReadImageData().** This is always a read-only value. It is dependent on the properties SIP_SCAN_MODE and SIP_SCAN_WIDTH_IN_PIXELS.

This property is used by the application to determine how large a buffer will be needed when calling **SI_ReadImageData().** The minimum buffer size should equal the number of lines requested multiplied by the value of SIP_LINE_WIDTH_IN_BYTES.

Item Types

SI_INT32

Container Types

SICON_SINGLE

SIP_LUT_BLUE

SIP_LUT_GREEN

SIP_LUT_GRAY

SIP_LUT_RED

Description

These four properties provide a means for setting a custom lookup table (LUT). Depending on the scanner model, any or all of these tables may be supported.

The format and item type of the lookup table depends on the scanner model. For example, the format could be an 8-bit table that provides 64K values used to map input values to output values. Or, it could be a 16-bit table that provides 256 values that act as input thresholds to determine the 8-bit output. See the model-specific info.

To disable a custom LUT, you must set its size to zero length. Custom LUTs are disabled by default.

Note that when any custom lookup table has been set with the properties SIP_LUT_RED, SIP_LUT_GREEN, SIP_LUT_BLUE, the following properties are ignored during a color scan:

SIP_GAMMA
SIP_HIGHLIGHT
SIP_SHADOW
SIP_BRIGHTNESS
SIP_CONTRAST
SIP_PHOTOMETRIC_INTERPRETATION

Those properties are also ignored during a grayscale scan if the SIP_LUT_GRAY property has been set.

Item Types

SI_UINT16
SI_UINT8

Container Types

SICON_ARRAY

SIP_MAX_SCAN_TIME_IN_SEC

Description

Specifies the maximum expected scan time in terms of seconds. This allows the application to set a maximum wait time. If the scanner has not responded in this time, then it is considered an error. This value is primarily provided to help support the WIA property WIA_DPS_MAX_SCAN_TIME.

Item Types

SI_INT32

Container Types

SICON_SINGLE

SIP_OPTICAL_RESOLUTION

Description

The native optical resolution of the scanner in terms of pixels per inch. This defines both the horizontal and vertical resolution.

Item Types

SI_INT32

Container Types

SICON_SINGLE

SIP_OPTICAL_WIDTH_IN_PIXELS

Description

The native optical width of the scanner in terms of pixels.

Item Types

SI_INT32

Container Types

SICON_SINGLE

SIP_PHOTOMETRIC_INTERPRETATION

Description

For binary scans, this property specifies whether a 0 indicates a white pixel or a black pixel. When set to SI_PI_BLACKZERO, 0 indicates a black pixel. Or in other words, when a black part of the image is scanned, it will produce a 0.

This property also works on grayscale and color scans by inverting the level ranges. For grayscale, setting SI_PI_WHITEZERO would cause pixels at level 255 to be black.

Item Types

SI_INT32

Container Types

SICON_SINGLE
SICON_LIST

Values Allowed

SI_PI_BLACKZERO
SI_PI_WHITEZERO

SIP_PLANARCHUNKY

Description

For color scanning, this property specifies whether the image data is returned in line planar or chunky format. In line planar format, an entire line of red pixels is returned, followed by and entire line of green pixels, and then an entire line of blue pixels. In chunky format, pixels are returned RGB-RGB-etc.

Planar:
```
RRRRRRRRRRRRRRR
GGGGGGGGGGGGGGG
BBBBBBBBBBBBBBB
```

Chunky:
```
RGBRGBRGBRGBRGBRGB
RGBRGBRGBRGBRGBRGB
```

(RGB order is shown here for illustration only. The actual channel order is set with the SIP_CHANNEL_ORDER property.)

If the scan mode is not set to color, then the value of this property does not apply.

Item Types

SI_INT32

Container Types

SICON_SINGLE
SICON_LIST

Values Allowed

SI_PC_PLANAR
SI_PC_CHUNKY

SIP_PREFEED_ENABLED

Description

Enables the prefeed feature. When prefeed is enabled and the paper is inserted (and triggers the paper sensor), the scanner will feed a short distance to "grab" the paper. This provides some tactile feedback to the user and ensures that the paper will be fed when the scan is started. But in some cases, when prefeed isn't desired, it can be disabled using this property.

Unlike most properties, the prefeed properties will persist after the interface is closed and the DLL is unloaded. (Settings are stored in the registry) This is because the prefeed feature is always in effect even when the application is not using the scanner interface.

Item Types

SI_BOOL

Container Types

SICON_SINGLE

Values Allowed

SI_TRUE
SI_FALSE

SIP_PREFEED_DELAY

Description

For the prefeed feature, this property allows you to set the time delay between when the paper is inserted (triggering the paper sensor) and when the paper is fed. Units are in milliseconds. Adjusting the prefeed delay to be longer gives the user more time to insert the paper before the scanner will "grab" it. After more practice at feeding in pages, the user may want to shorten the delay.

The setting for this property only applies when prefeed is enabled. (See the SIP_PREFEED_ENABLED property.) Unlike most properties, the prefeed properties will persist after the interface is closed and the DLL is unloaded. (Settings are stored in

the registry) This is because the prefeed feature is always in effect even when the application is not using the scanner interface.

Item Types

SI_INT32

Container Types

SICON_RANGE

SIP_PREFEED_DISTANCE

Description

For the prefeed feature, this property allows you to adjust how far in the paper gets fed. The distance is specified in terms of hundredths of an inch. If you find that too much of the top of the page is being clipped off, you can shorten this distance. Longer distances will feed the paper in further.

The setting for this property only applies when prefeed is enabled. (See the SIP_PREFEED_ENABLED property.) Unlike most properties, the prefeed properties will persist after the interface is closed and the DLL is unloaded. (Settings are stored in the registry) This is because the prefeed feature is always in effect even when the application is not using the scanner interface.

Item Types

SI_INT32

Container Types

SICON_RANGE

SIP_SCAN_LENGTH_IN_LINES

Description

Set the length of the scan in terms of lines in the currently set vertical resolution.

The number of lines is specified in the current vertical resolution. If the vertical resolution is set to 100 DPI and SIP_SCAN_LENGTH_IN_LINES is set to 100 lines, then the scan length will be 1 inch. But if the resolution is changed to 300 DPI, then the scan length will be 1/3 of an inch.

Item Types

SI_INT32

Container Types

SICON_SINGLE
SICON_RANGE

SIP_SCAN_MODE

Description

Sets the mode of the scan, such as color or grayscale. The possible scan modes available depend on the scanner model.

Item Types

SI_INT32

Container Types

SICON_SINGLE
SICON_LIST

Values Allowed

SI_SCANMODE_BW
SI_SCANMODE_GRAY
SI_SCANMODE_COLOR

SIP_SCAN_RATE

Description

Allows adjustment of the scanning speed in inches per second. By default, scans will run at the fastest speed possible for a give mode and resolution.

This property is optional and need not be set. But in some cases, you may want to purposely slow the scanning speed to avoid scanner pauses which can occur if the computer cannot read data as fast as the scanner is providing it. This can happen for very slow computers or if the connection is the slower USB 1.1. However, scanner pauses are not an error and the full page will still be scanned.

When you retrieve this property, the current value returned is the speed that paper will move in *inches per second* for the current scan mode, resolution, and duplex settings. Since this property uses a range container, you'll also get a minimum and maximum value that this property can be set to. Setting the current value lower than the maximum will result in slower scan rates.

This property depends on the current scan mode, resolution, and duplex settings (for duplex scanner models). Changing any one of those will cause the current scan rate value to change to reflect the new scan speed. Therefore, you should re-read the value after changing any of those properties.

Adjusting this property will change the *scan* speed but not the feed speed. Thus if you scan a page with a top offset, the page will first be fed a distance equal to the top offset at the *feed rate*. That feed rate can be controlled by the SIP_FEED_RATE property. Then the scanning portion proceeds at the speed set by the SIP_SCAN_RATE property.

The SIP_SCAN_RATE property may or may not be supported depending on the scanner model.

Item Types

SI_FLOAT32

Container Types

SICON_RANGE

SIP_SCAN_WIDTH_IN_PIXELS

Description

Set the width of the scan in terms of pixels in the currently set horizontal resolution.

The number of pixels is specified in the current vertical resolution. If the horizontal resolution is set to 100 DPI and SIP_SCAN_WIDTH_IN_PIXELS is set to 100, then the scan width will be 1 inch. But if the horizontal resolution is changed to 300 DPI, then the scan width will be 1/3 of an inch.

Item Types

SI_INT32

Container Types

SICON_SINGLE
SICON_RANGE

SIP_SHADOW

Description

Sets the shadow level. The shadow level is the minimum black level. So for an 8-bit grayscale image, for example, this would normally be set to 0 by default. If this value is increased to a higher value, then image levels that were originally brighter will appear darker.

Note the interaction between SIP_HIGHLIGHT and SIP_SHADOW when setting the current value. These two properties together define the range from dark to light. SIP_SHADOW can never be equal to or higher than the value of SIP_HIGHLIGHT. So changing the current value of SIP_SHADOW may cause the current SIP_HIGHLIGHT to be automatically adjusted to be one greater than SIP_SHADOW if it was originally less than or equal to it. You can check the current value of SIP_HIGHLIGHT to confirm this. Likewise, changing the current value of SIP_HIGHLIGHT may affect the current value of SIP_SHADOW.

When any custom lookup table has been set with the properties SIP_LUT_RED, SIP_LUT_GREEN, SIP_LUT_BLUE, or SIP_LUT_GRAY, the shadow value set in this property is ignored.

Item Types

SI_INT32

Container Types

SICON_RANGE

SIP_SPOOLER_ENABLED

Description

Enables the scan spooler. The spooler is enabled by default. When a scan is started, the spooler reads the scan data from the scanner and stores to a file on disk. This prevents the scanner from pausing in the middle of a scan if its buffer becomes full.

If the scanner supports duplex (the SIP_DUPLEX_ENABLED property exists) and duplex is enabled, then the spooler must also be enabled in order to read both sides. If the spooler is disabled, then only once side or the other can be read. If the spooler is explicitly disabled by the application, it will not become automatically enabled when duplex is enabled. It is up to the application to re-enable the spooler.

Item Types

SI_BOOL

Container Types

SICON_SINGLE

Values Allowed

SI_TRUE
SI_FALSE

SIP_THRESHOLD

Description

Sets the threshold level for binary scans. The default value for 8-bit binary is 128.

This property is ignored grayscale and color scan modes.

Item Types

SI_INT32

Container Types

SICON_RANGE

SIP_USB_RATE

Description

Reports the current USB connection rate, either FULL (12Mbits/sec.) or HIGH (480Mbits/sec.). This property is read-only. It is optional and may not exist for all scanner models.

If the host adapter is USB 1.1 or if there is a USB 1.1 hub anywhere along the chain, then this property's current value will return `SI_USB_FULL`. Otherwise it will return `SI_USB_HIGH`. This can be useful for determining if you need to slow down the scan rate to prevent scanner pauses.

The connection rate is determined by the scanner itself as a result of negotiation and not by an actual measurement of transfer speed. To get an actual measurement of the USB transfer speed, see the **SI_Diagnostic()** API function.

Item Types

SI_INT32

Container Types

SICON_LIST

Values Allowed

SI_USB_FULL
SI_USB_HIGH

SIP_USB_SERIAL_NUMBER

Description

Reports the scanner's USB serial number as a zero-terminated ASCII string. This property is read-only. It is optional and may not exist for all scanner models.

If this string is empty (length of zero), then the scanner does not have a USB serial number.

Item Types

SI_STR

Container Types

SICON_SINGLE

SIP_XOFFSET

Description

Sets the X offset (left margin) of the scan window in terms of pixels in the current resolution.

Increasing the X offset reduces the allowable range for the SIP_SCAN_WIDTH_IN_PIXELS property.

Item Types

SI_INT32

Container Types

SICON_RANGE

SIP_XRESOLUTION

Description

Sets the horizontal resolution in terms of pixels per inch.

In most scanners, the X and Y resolutions must be the same. Therefore, changing SIP_XRESOLUTION will result in SIP_YRESOLUTION changing to match.

Item Types

SI_INT32

Container Types

SICON_RANGE
SICON_LIST

SIP_YOFFSET

Description

Sets the Y offset (top margin) of the scan window in terms of pixels in the current resolution.

Increasing the Y offset reduces the allowable range for the SIP_SCAN_LENGTH_IN_LINES property.

Item Types

SI_INT32

Container Types

SICON_RANGE

SIP_YRESOLUTION

Description

Sets the vertical resolution in terms of pixels per inch.

In most scanners, the X and Y resolutions must be the same. Therefore, changing SIP_YRESOLUTION will result in SIP_XRESOLUTION changing to match.

Item Types

SI_INT32

Container Types

SICON_RANGE
SICON_LIST