Development Using Microsoft .NET Languages

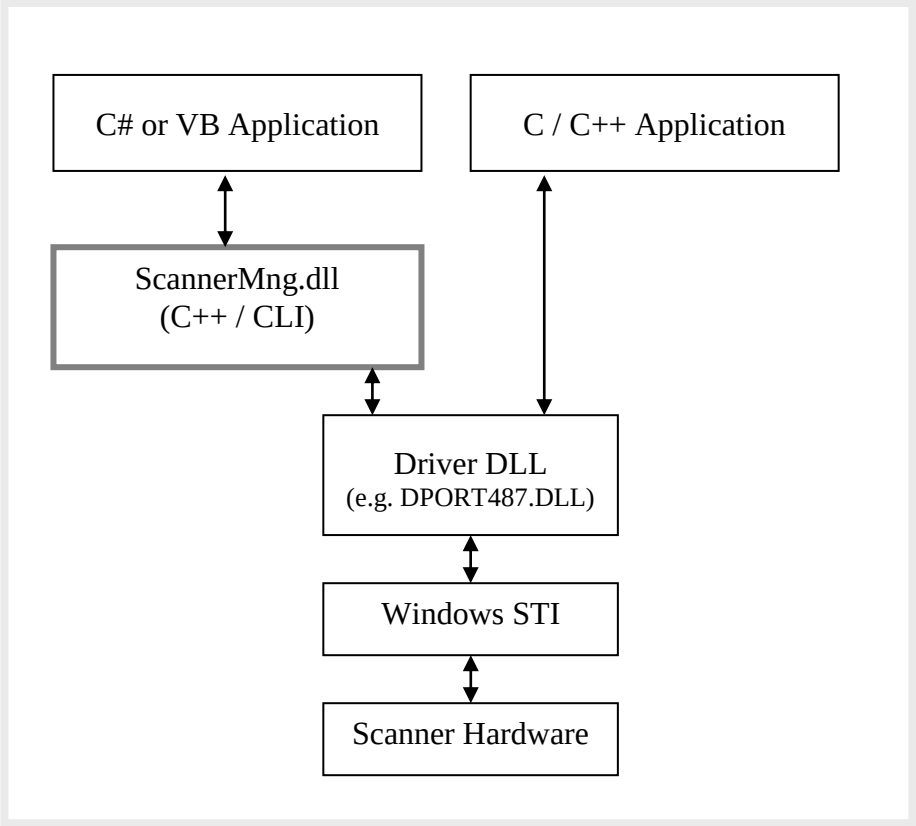
Introduction

Document Capture Technologies DocketPORT scanners support applications developed using Microsoft .NET-based languages. This is accomplished through two different methods, either one of which may be used. The first method is through a thin translation layer implemented in `ScannerMng.dll`. The second method is by using the P/Invoke mechanism to make calls to the low-level driver DLL.

ScannerMng.dll

`ScannerMng.dll` is an interface layer that translates from managed code to unmanaged code, and thus P/Invoke methods are not needed. This DLL can be renamed as desired. Source code for `ScannerMng.dll` is provided. It is implemented in C++/CLI and therefore requires Visual Studio 2005 to make any modifications.

The implementation of this managed interface is intended to parallel that of the unmanaged API for consistency. Most of the information in the main Scanner Development Kit document still applies and is also used as a basis for developing managed applications. Therefore it's important to read the main SDK document even when using managed code. Only the differences will be described here.



While the API functions are similar, you'll see that parameters are handled in a way that's more appropriate for a managed language. For example, where in C or C++ you would pass a text string parameter as a char*, in C# you would pass a string object.

P/Invoke

P/Invoke (short for Platform Invoke) is Microsoft's generalized mechanism for translating from managed code to unmanaged code. (See Using P/Invoke to Call Unmanaged APIs from Your Managed Classes on the MSDN website) The DCT driver functions can be accessed through P/Invoke.

The remainder of this document will describe only the `ScannerMng.dll` method and not P/Invoke. However, two small sample applications are provided, one for C# (`CSSamplePI`) and one for Visual Basic (`VBSamplePI`), that demonstrate the use of P/Invoke. You can cut and paste the function declarations into your own code. Please see the section on sample applications below.

Requirements

Development of .NET programs that access the scanner driver requires Visual Studio 2005 Professional Edition or later. (Other editions of Visual Studio 2005 may work but are not tested or supported) There are a couple of reasons for this In the case of using `ScannerMng.dll`, since this DLL is built using Visual Studio 2005, projects in earlier versions of Visual Studio cannot add this DLL as a reference. In the case of using P/Invoke, the driver functions use the `cdecl` calling convention which is not supported in earlier versions of Visual Studio.

Also note that the version 8.0 runtime libraries are required for `ScannerMng.dll`. This includes files:

MSVCR80.DLL
MSVCM80.DLL
MSCOREE.DLL

These libraries cannot be linked statically.

Sample Applications

A small sample application is provided in both C# (`CSSample`) and Visual Basic (`VBSample`) format. This program scans a letter-size page and saves it to BMP-format file. This demonstrates

the basics of scanning, although it doesn't exercise all of the possible scanner capabilities. A sample program is often the fastest and easiest way to see how the scanner API works.

Also included are sample applications `CSSamplePI` and `VBSamplePI` which are the equivalent of the programs `CSSamplePI` and `VBSamplePI` except that they use the P/Invoke method. As you'll notice, the code for the P/Invoke samples is almost the same. The notable differences are:

- With the P/Invoke samples, you don't create an instance of the Scanner class. Instead, you call the API functions as static functions with the class name qualifier. For example, `SIScanner.SI_StartScan()` instead of `m_scanner.SI_StartScan()`.

- In the SIProperty class, some fields that are intended to be (pointers to) arrays in C++ cannot be accessed as arrays in managed code. For example, the field `SIProperty.list.items.piVal` is a pointer to an array of integers. In these cases, the array is just copied to a managed array. In C# this would be:

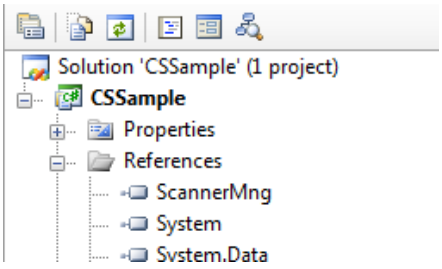
```
 // scanModes will have the list
Int32[] scanModes = new Int32[prop.list.numItems];
Marshal.Copy(prop.list.items.piVal, scanModes, 0, prop.list.numItems);
```

  See the sample applications for examples of this.

As mentioned earlier, the remainder of this document will refer to only the `ScannerMng.dll` method. But reference to the sample applications should make the P/Invoke method fairly straightforward.

Developing With the Managed API

To begin building a managed application, add a reference to the `ScannerMng.dll` file.



At the top of source files that use the scanner API add the following line to specify the namespace "`ScannerMng`".

```
using ScannerMng;
```

The SIScanner class

In the managed interface, all scanner function calls are methods of the SIScanner class. So in order to call the functions you first create an instance of SIScanner.

```
SIScanner scanner = new SIScanner("DPORT487.dll");
```

This is different from the unmanaged interface which, being a C interface, does not use a SIScanner class.

The constructor takes one parameter which is the filename of the scanner DLL. The name used here is the "`DPORT487.dll`" which is the DLL filename for the DocketPORT 487. But you should use the filename for your particular scanner model. If you look at the diagram in the introduction section, you can see that the `ScannerMng.dll` file must in turn load the `DPORT487.dll` file, so it needs to know the filename. Since it loads using the Win32 function `LoadLibrary()`, the same DLL searching rules apply as with `LoadLibrary()`. (See Microsoft documentation for more information on `LoadLibrary()`) Typically the `DPORT487.dll` file is installed to the System32 directory so you don't need to specify a full pathname.

The `SIScanner` constructor may throw an `ApplicationException` exception if the specified DLL cannot be loaded. This is the only exception that the `SIScanner` constructor might throw. No other function in the API will throw exceptions; all other functions return a value of type `SIResult` to indicate success or failure. This is consistent with the unmanaged version of the API.

Open the Interface

The first function you must call is `SI_OpenInterface()`. This initializes the interface and must be called before any other functions. This is the same as with the unmanaged interface except that the parameter is a `string` instead of a `char` pointer. The parameter is the textual name of the scanner, which depends on the scanner model.

```
SIResult result;

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

Note that the return value `SIResult` is an enumeration. All possible return values are members of the `SIResult` enumeration. So you could test those for success using code something like the following:

```
if (result != SIResult.SIR_SUCCESS)
{
      string str = "";
      scanner.SI_GetLastErrorText(ref str);
      MessageBox.Show(str);
      return;
}
```

This example also makes use of the `SI_GetLastErrorText()` function, which returns a text string describing the last result returned from an API function.

All constant values that are defined as either `const` or `#define` in the unmanaged API are defined as members of some enumeration or class in the managed API. All constant identifiers are the same as in the unmanaged version (ScannerAPI.h file) except that they must be preceded by the qualifier.

Two notable qualifiers in the ScannerMng namespace are:

- SIResult. Contains constants returned from functions as return values.
  For example:
  SIResult.SIR_SUCCESS
  SIResult.SIR_INTERFACE_NOT_OPEN

- SIProperty. Contains constants used in setting and getting scanner properties using the SIProperty class.
  For example,
  SIProperty.SIP_OPTICAL_RESOLUTION
  SIProperty.SI_INT32
  SIProperty.SI_TRUE

There are also other enumerations used for specific functions. Refer to the function reference for more info.

Calibrate the Scanner

If this is the first time the scanner is being used, it has probably not been calibrated yet. In that case you need to perform calibration. This is handled in a way similar to the unmanaged interface. The `CSSample.cs` sample program has an example of calibrating the scanner. Please refer to the sample code and also the function reference for more detail.

One notable difference in the managed API is that the callback function used to report progress is a delegate instead of a function pointer. This is an optional parameter depending on whether you'd like to get calibration progress and cancel the operation.

The following code is an example of calibration.

```
SIResult result;
SICalibrationState calState = SICalibrationState.NOT_CALIBRATED;

result = scanner.SI_IsCalibrated(ref calState);

if (result == SIResult.SIR_SUCCESS)
{
      if (calState == SICalibrationState.NOT_CALIBRATED)
      {
            // 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 = scanner.SI_Calibrate(
                        SICalibrationTarget.SI_CT_DEFAULT, null);

            if (result == SIResult.SIR_SUCCESS)
            {
                  MessageBox.Show("Calibration was successful.");
            }
            else
            {
                  MessageBox.Show("Calibration failed.");
            }
      }
}
```

Properties

Scanner properties work in a way very similar to the unmanaged API. You set the scanner properties using the `SI_SetProperty()` function and you query the properties using the `SI_GetProperty()` function. Both of these functions take a single parameter of type `SIProperty`.

However the main difference in the managed version is that the `SIProperty` parameter is an class instead of a structure and it includes members that are allocated from the managed heap. So to retrieve the property for X resolution, you would write:

```
SIProperty prop = new SIProperty();       // allocate a SIProperty object
SIResult result;

prop.propertyID = SIProperty.SIP_XRESOLUTION;   // specify the ID of the
                                                //    property we want
result = scanner.SI_GetProperty(prop);          // Get the property

Debug.Assert(result == SIResult.SIR_SUCCESS);   // make sure it succeeded

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

For consistency with the unmanaged API, the `SIProperty` class is implemented to be similar to the unmanaged structure which actually contains some unions. Although unions are not supported in C#, the layout and meaning of the fields are still similar.

Referring the example above, you can access `prop.`*`list`*`.current.iVal` because the `SIP_XRESOLUTION` property uses a list container (as specified in the `prop.containerType` field). The `list` container is allocated during the call to `SI_GetProperty`. It would be invalid to try to access `prop.range` or `prop.single`.

The same case exists with the item type. In the example above, since the item type for `SIP_XRESOLUTION` (specified in the `prop.containerType` field) is SI_INT32, you would access this value using `prop.list.current.`*`iVal`*`.` It would be invalid to try to access the member `fVal` or `strVal`.

To set a given property, the simplest (and recommended) way is to call SI_GetProperty first so that all fields of the `SIProperty` object are correctly filled out. Then you can modify the "current" value and pass the `SIProperty` object to `SI_SetProperty()` to set it.

The following example sets the X resolution property to 200 DPI.

```
SIProperty prop = new SIProperty();
SIResult result;

prop.propertyID = SIProperty.SIP_XRESOLUTION;   // initialize the ID
result = scanner.SI_GetProperty(prop);          // Get the property
Debug.Assert(result == SIResult.SIR_SUCCESS);   // make sure it succeeded

prop.list.current.iVal = 200;                   // specify the new current
                                                //    resolution
result = scanner.SI_SetProperty(prop);          // Set the property
Debug.Assert(result == SIResult.SIR_SUCCESS);   // make sure it succeeded
```

Of course, it's also possible to set a property without calling `SI_GetProperty` first. You can setup the required four fields of the `SIProperty` class. But this method is longer, even longer in managed code than in unmanaged code because you also need to allocate reference values within the class as well. For example,

```
SIProperty prop = new SIProperty();
SIResult result;

prop.propertyID = SIProperty.SIP_XRESOLUTION;   // initialize the ID
prop.containerType = SIProperty.SICON_LIST;     // uses a list container
prop.itemType = SIProperty.SI_INT32;       // Set the item type to 32-bit
                                           //          integer
prop.list = new SIList();                  // allocate a list container
prop.list.current.iVal = 200;              // specify the new current
                                           //    resolution
result = scanner.SI_SetProperty(prop);     // Set the property
Debug.Assert(result == SIResult.SIR_SUCCESS);   // make sure it succeeded
```

The difference between this and the unmanaged code version is that you also need to allocate the list container. However, you do not need to allocate the list of possible items (`prop.list.items.piVal`) since the list is ignored by `SI_SetProperty`.

As described in the SDK manual, you can also use a `SICON_SINGLE` container to set the current value of properties that have containers of types `SICON_LIST` or `SICON_RANGE`. Please refer to the SDK manual for more information about using scanner properties.

Check for Paper

Before starting a scan, you'll normally want to check whether or not there is paper inserted. The following C# example displays a message prompting the user to insert the paper and confirm that paper is in.

```
SIResult result;
SIPaperStatus paperStatus = SIPaperStatus.SI_PS_PAPER_OUT;

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

while (paperStatus == SIPaperStatus.SI_PS_PAPER_OUT)
{
      DialogResult dlgRes = MessageBox.Show(
            Insert paper and press any key to begin the scan.”,
            "", MessageBoxButtons.OKCancel);

      if (dlgRes == DialogResult.Cancel)
      {
            // skip the scan
            return;
      }

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

      if(result != SIR_SUCCESS)
      {
            MessageBox.Show("Error getting paper status.");
      }
};
```

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. See the main SDK document for more detail.

```
SIResult result;

result = scanner.SI_StartScan();

Debug.Assert(result == SIResult.SIR_SUCCESS);
```

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

```
SIResult result;
uint numLinesReturned = 0;

// Allocate a buffer big enough to hold 10 lines
// The widthInBytes variable contains the width of one line in bytes
// as obtained from the SIP_LINE_WIDTH_IN_BYTES property.
byte[] lineBuffer = new byte[widthInBytes * 10];

result = scanner.SI_ReadImageData(
                  lineBuffer,
                  10,         // Read at most 10 lines
                  0,          // specify the front page (if duplex)
                  ref numLinesReturned);  // num lines actuall returned
```

End the Scan

When you are done reading image data, call `SI_EndScan()` to end the scan for this page. You can then call `SI_FeedPaperOut()` to feed the paper all the way out of the scanner.

```
// End the page.
scanner.SI_EndScan();

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

Close the Interface

Once you have completely finished scanning, you must call SI_CloseInterface() to clean up any resources.

```
// Close the scanner interface.
scanner.SI_CloseInterface();
```

Function Reference

The scanner is controlled through a set of member functions of the SIScanner class. The functions are callable from a managed language such as C#.

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

```
public enum SIResult;
```

The result code normally reports the success or failure of the function. Possible result codes are members of `SIResult` and begin with the prefix "`SIR_`". In general, the code `SIR_SUCCESS` is returned when a function completes successfully.

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(
      string pName
);
```

Parameters

pName	[in] A case-sensitive, ASCII string specifying the scanner to open. This string is product-specific. Contact DCT for the string to use.

Return Values

SIResult.SIR_SUCCESS	The scanner was opened successfully.

SIResult.SIR_ALREADY_OPEN	SI_OpenInterface has already been called and the scanner interface is already open.
SIResult.SIR_UNKNOWN_MODEL_NAME	The string passed in with the pName parameter was not recognized as a supported model.
SIResult.SIR_BAD_PARAMETER	pName was NULL.
SIResult.SIR_SCANNER_NOT_READY	The scanner hardware could not be detected. The scanner is not connected.
SIResult.SIR_INITIALIZATION_FAILURE	The scanner could not be opened and initialized.
SIResult.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

SIResult.SIR_SUCCESS	The scanner interface was closed successfully.
SIResult.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(
     ref SICalibrationState pState
);
```

Parameters

pState	[out] On successful return, pState is set to `SICalibrationState.CALIBRATED` if the scanner is calibrated, `SICalibrationState.NOT_CALIBRATED` if the scanner is not calibrated.

Return Values

SIResult.SIR_SUCCESS	The calibration state was returned successfully. The value pointed to by pState has been set.
SIResult.SIR_BAD_PARAMETER	The *pState* parameter was NULL.
SIResult.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(
      SICalibrationTarget target,
      ProgressCallbackDelegate progressCallback
);
```

Parameters

target [in] One of the SICalibrateTarget enumeration values 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.

progressCallback [in] A delegate 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

SIResult.SIR_SUCCESS	The scanner was calibrated successfully.
SIResult.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.
SIResult.SIR_NOT_CALIBRATED	The scanner could not be calibrated.
SIResult.SIR_BAD_PARAMETER	The value of the *target* parameter was not valid.
SIResult.SIR_USER_CANCELLED	A callback function was specified in *progressCallback* and that callback returned `SICallbackResult.CANCEL` to cancel the calibration process.
SIResult.SIR_FILESYSTEM_ACCESS_DENIED	The file system did not allow access to the calibration file.

SIResult.SIR_SCANNER_BUSY	The scanner is busy and cannot perform calibration.
SIResult.SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.
SIResult.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 *progressCallback* 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 *progressCallback* parameter is not null, it must point to a delegate that will be called during the calibration process to report the percentage complete. The delegate function must have the following form:

```
public SICallbackResult ProgressCallback(UInt32 percentComplete)
```

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 `SICallbackResult.CONTINUE`. If you return `SICallbackResult.CANCEL`, the calibration will be aborted and the **SI_Calibrate** function will return `SIResult.SIR_USER_CANCELLED`.

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(
      ProgressCallbackDelegate progressCallback
);
```

Parameters

progressCallback	[in] A delegate 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

SIResult.SIR_SUCCESS	The scanner completed cleaning successfully.

SIResult.SIR_USER_CANCELLED	A delegate function was specified in *progressCallback* and that function returned `SICallbackResult.CANCEL` to cancel the cleaning process.
SIResult.SIR_SCANNER_BUSY	The scanner is busy and cannot perform cleaning.
SIResult.SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.
SIResult.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 *progressCallback* 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 *progressCallback* parameter is not null, it must point to a delegate function that will be called during the cleaning process to report the percentage complete. The function must have the following form:

```
public SICallbackResult ProgressCallback(UInt32 percentComplete)
```

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 `SICallbackResult.CONTINUE`. If you return `SICallbackResult.CANCEL`, the cleaning will be aborted and the **SI_Clean** function will return `SIResult.SIR_USER_CANCELLED`.

SI_GetScannerStatus

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

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

Parameters

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

SIScannerStatus.SI_SS_OFFLINE	The scanner is not available.
SIScannerStatus.SI_SS_READY	The scanner is online and ready to scan.
SIScannerStatus.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

SIResult.SIR_SUCCESS	The scanner status was obtained successfully.
SIResult.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(
     ref SIPaperStatus paperStatus
);
```

Parameters

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

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

Return Values

SIResult.SIR_SUCCESS	The scanner status was obtained successfully.
SIResult.SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called. The paper status cannot be determined.
SIResult.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,
      ref SIButtonStatus buttonStatus
);
```

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.

buttonStatus [out] A reference 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:

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

Return Values

SIResult.SIR_SUCCESS	The scanner status was obtained successfully.
SIResult.SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called. The button status cannot be determined
SIResult.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 property
);
```

Parameters

property	[in-out] A SIProperty object that receives the property information. The propertyID member of this class must be set to the ID of the property to retrieve.

Return Values

SIResult.SIR_SUCCESS	The property information was obtained successfully.
SIResult.SIR_BAD_PARAMETER	The *property* parameter was null.
SIResult.SIR_PROPERTY_UNSUPPORTED	The property ID set in *property* was not one of the support IDs for this scanner.
SIResult.SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.

Remarks

Upon successful completion of the function, the *property* parameter will be filled out with relevant information about that property, including the current value and valid values.

See the documentation elsewhere 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 property
);
```

Parameters

property	[in] An SIProperty object. Four fields of the property must be set: • *propertyID* • *containerType* • *itemType* • the *current* value (within the appropriate container)

Return Values

SIResult.SIR_SUCCESS	The scanner status was obtained successfully.
SIResult.SIR_BAD_PARAMETER	The *property* parameter was null.
SIResult.SIR_PROPERTY_UNSUPPORTED	The property ID set in the *property* parameter was not one of the support IDs for this scanner.
SIResult.SIR_PROPERTY_INVALID_VALUE	One or more of the fields in *property* were invalid or out of range. Use SI_GetProperty to find valid values.
SIResult.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 documentation elsewhere 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

SIResult.SIR_SUCCESS	The scan was initiated status was obtained successfully.
SIResult.SIR_SCANNER_BUSY	The scanner is busy and cannot scan. Another process may currently be scanning.

SIResult.SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.
SIResult.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(
      Byte[] buffer,
      UInt32 numberOfLinesToRead,
      UInt32 pageNumber,
      ref UInt32 numberOfLinesReturned
);
```

Parameters

buffer	[out]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 reference 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

SIResult.SIR_SUCCESS	The function returned successfully with zero or more lines in the buffer.
SIResult.SIR_BAD_PARAMETER	One or more of the parameters were invalid.
SIResult.SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.
SIResult.SIR_ENDOFDATA	There is no more data for the specified page.
SIResult.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.

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

SIResult.SIR_SUCCESS	The scan was ended successfully.
SIResult.SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.
SIResult.SIR_SCANNER_BUSY	The scanner is busy and cannot feed. Another process may currently be scanning.
SIResult.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

SIResult.SIR_SUCCESS	The function performed the feed successfully.
SIResult.SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called. The paper status cannot be determined.
SIResult.SIR_SCANNER_BUSY	The scanner is busy and cannot feed. Another process may currently be scanning.
SIResult.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

SIResult.SIR_SUCCESS	The function performed the feed successfully.
SIResult.SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called. The paper status cannot be determined.

SIResult.SIR_SCANNER_BUSY	The scanner is busy and cannot feed. Another process may currently be scanning.
SIResult.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,
      SIDiagInfo  diagInfo
);
```

Parameters

test	[in] An ID value that specifies which test to perform. See the Remarks section for a list of possible values.
diagInfo	[in] An object if type derived from SIDiagInfo. The actual type of the object passed in depends which test is specified in the *test* parameter. This object may contain both input and output values for the diagnostic test. See the Remarks section for more info.

Return Values

SIResult.SIR_SUCCESS	The test result was returned successfully.
SIResult.SIR_BAD_PARAMETER	The test specified was unknown or the diagInfo parameter was null.
SIResult.SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.
SIResult.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 type of the second parameter d*iagInfo* depends on the test being performed. The following table lists the valid tests types which must be passed in for each test. The comments also describe any members which must be set up before calling the function, as well as result values returned in members.

test	*dialogInfo*	Comment
`SIDiagTest.SI_DIAG_HW_PRESENCE`	`SIDiagHwPresenceInfo`	**Output**: the result of the test is returned in the `result` member variable. Possible results are: • `SIDiagResult.SI_DIAG_PASS` – the scanner is connected. • `SIDiagResult.SI_DIAG_FAIL` – the scanner is not connected.
`SIDiagTest.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 a C# example of calling SI_Diagnostic() to do a Read Performance test.

```
SIDiagReadPerfInfo rpi = new SIDiagReadPerfInfo();

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

// Do the test.
m_scanner.SI_Diagnostic(SIDiagTest.SI_DIAG_READ_PERFORMANCE, rpi);

// Display the result.
MessageBox.Show("The time to read " +
            rpi.readLengthInKB + " KB was " +
            rpi.timeInMicroSec + " us.");
```

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. Applications will not normally call it. If you're writing an application, you can ignore this function.

```
SIResult SI_GetEvent(
      SIEventInfo eventInfo
);
```

Parameters

eventInfo	[in-out] An SIEventInfo object. The *eventDataSize* and *eventData* structure fields must be filled out before passing it to SI_GetEvent. See Remarks for more detail.

Return Values

SIResult.SIR_SUCCESS	The event code was recognized and successfully converted to test. The *eventName* field of the e*ventInfo* structure was filled with the event name.
SIResult.SIR_BAD_PARAMETER	The e*ventInfo* parameter was null.
SIResult.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 an SIEventInfo object. Its structure is defined essentially as:

```
public ref class SIEventInfo
{
	/// [in] The size in bytes of the data in the eventData array. (MAX 20)
	Byte   eventDataSize;

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

	/// [out] Bit flags that indicate which event(s) occurred.
	UInt32 eventFlags;

};
```

The maximum size of the eventData array is 20 bytes.

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 defined in SIEventInfo and are:

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

SI_Reset

The SI_Reset function resets the scanner to a known state.

```
SIResult SI_Reset();
```

Parameters

None.

Return Values

SIResult.SIR_SUCCESS	The scanner was reset successfully.
SIResult.SIR_INTERFACE_NOT_OPEN	The scanner interface is not open. SI_OpenInterface was not called.
SIResult.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_GetLastErrorText.

See Also

SI_GetLastErrorText

SI_GetLastErrorText

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

```
SIResult SI_GetLastErrorText(ref string errorText);
```

Parameters

errorText	[out] A reference to a string which upon return will describe the last API result that occurred.

Return Values

SIResult.SIR_SUCCESS	The text string was returned successfully.

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