TwainScanning

About

TwainScanning is a .NET library that enables you to acquire images from any device that has TWAIN drivers and convert acquired images to various formats including bitmap, jpeg, tiff and pdf.

Installation

Download file from the following link http://twainscanning.net/free-download and extract files from the archive.

Setting up the project

- Add reference TwainScanning.dll into your project.
- Set "Solution Platform" to "x86"
    - Use whenever possible, provides the best performance and all features
- If the project needs to be AnyCPU or x64, `Bridgex86` should be used
    - `Bridgex86` is a reliable alternative, but it provides a different workflow with a more limited feature set and a slight performance impact

Providing the license key

For evaluation purposes, one can skip this step. If it is skipped all functionality will remain but acquired images will have watermark written on them.

When you buy the license, you will receive serial key valid only for your company. To provide the license key to the library call the `GlobalConfig.SetLicenseKey` function.

Using the library

Initializing data source manager (TWAIN)

Instantiate an instance of the `DataSourceManager` providing them your project's main form and additional info about your application.

By instantiating `DataSourceManager` TWAIN interface (source manager) is opened and initialized, the drivers are loaded and ready to use.

```
var dsm = new DataSourceManager(this); // opens TWAIN inside form
```

If the TWAIN is not supported by the operating system or source manager cannot be opened, instantiation will throw exception and object will not be created.

By disposing of this object TWAIN interface will close, and resources will be freed, so don't forget to dispose it.

Choosing the data source (scanner)

There are two ways to select scanner: from code or by displaying TWAIN built in GUI to choose the source.

From code

List of all available scanners, as a list of the `TwIdentity` objects, can be obtained through the `DataSourceManager.AvailableSources()` method. Use one of the objects from the list to open specific data source (scanner).

```
//listing and writing scanner names
List<TwIdentity> scannerIds = dsm.AvailableSources();
foreach (TwIdentity id in scannerIds)
       Console.WriteLine(id.ProductName);
```

By built-in GUI

Call the `DataSourceManager.SelectDefaultSourceDlg()` method to show dialog on which one can select default data source (scanner).

```
dsm.SelectDefaultSourceDlg();//Setting the default scanner by default twain
```

Opening the data source (scanner)

Data source (scanner) is opened by calling `DataSourceManager.Open()`. If as an argument is passed device name or an instance of the `TwIdentity` object, specific data source will be opened. Otherwise the default data source will be opened.

```
var ds = dsm.OpenSource(); // opens default source
var ds = dsm.OpenSource("ScannerXYZ"); // opens default source ScannerXZY
```

This function returns the `DataSource` object which represents opened, initialized and useful data source (scanner). If it fails to open (for example if a scanner is turned off or it is already opened), the function will throw.

By disposing `DataSource` object the data source (scanner) will be closed and resources freed. If one forgets to dispose this object's data source, it may remain in opened state. That may prevent opening this scanner again.

Basic settings

The `DataSource` object exposes some of the most commonly used setting for easy access through these fields as aliases: `PageSize`, `ColorMode`, `PixelDepth`, `Resolution`, `UseFeeder`, `UseDuplex` and `TransferMechanism`.

```
ds.Resolution.Value = 200f;                //setting resolution
ds.TransferMechanism.Value = TwSX.Memory;  //setting transfer mechanism
float resolution = ds.Resolution.Value;    //getting resolution
TwSX mech = ds.TransferMechanism.Value;    //getting transfer mechanism
```

Setting and getting values on these fields is accomplished through the `Value` property.

```
float[] resolutions = ds.Resolution.AvailableValues; //supported resolutions
TwSX[] mechs = ds.TransferMechanism.AvailableValues; //supported mechanism
```

Finding all available values on these fields is accomplished through the `AvailableValues` property.

All settings (capabilities)

All capabilities of the data source (scanner) can be accessed through the `Settings` object on the `DataSource`. Capabilities are arranged into the categories. For a full list of categories and containing capabilities see (Appendix: List of capabilities).

The value of specific capability can be obtained through the Value property on the capability object. If a capability is read-only Value setter is not implemented. If a capability can have multiple values, the Value property is an array.

All supported values can be retrieved through `SupportedValues` property.

`IsSupportedOnThisDevice()` method returns if this opened data source supports this capability.

```
var capOrientation = ds.Settings.ImageAcquire.Orientation; // info about orientation
if (capOrientation.IsSupportedOnThisDevice()) // can scanner support orientation?
{
    TwOR orientation = capOrientation.Value;
    Console.Write(" Current orientation is " + orientation);

    Console.Write(" All possible orientations are:");
    foreach(TwOR orient in capOrientation.SupportedValues)
        Console.Write(" " + orient);

    capOrientation.Value = TwOR.Landscape; //set orientation to Landscape
}
```

Acquiring

There are two ways of scanning available: synchronous and asynchronous.

Synchronous scanning

Synchronous scanning starts by calling `DataSource.Acquire(…)` method. This method doesn't return until scanning is finished.

It is the recommended way for scanning in console applications.

```
ImageCollector coll1 = ds.Acquire(false);// scanning has started (no scanner gui)
coll1.SaveAllToBitmaps(...) ;//scanning has finished and collector is ready

var coll2 = new ImageCollectorPdf(@"C:\someDir\someFile.pdf");
ds.Acquire(coll2, false);// scanning has started (no scanner gui)
coll2.Dispose(); // scanning has finished ()
```

Asynchronous scanning

Asynchronous scanning starts by calling `DataSource.AcquireAsync(…)` method. This method returns immediately, and information about scanning status is provided using events only (see Events chapter).

It is the recommended way for scanning in non-console applications.

```
// using general collector
private void Scan(){
    ds.AcquireAsync(onFinishedScanning, true); // scanning has started
}

private void onFinishedScanning(ImageCollector collector){ // scanning has finished
    collector.SaveAllToMultipageTiff(@"C:\someFolder\someFile.tiff");
}

// using specific collector
private void Scan(){

    var collector = new ImageCollectorPdf(@"C:\someFolder\someFile.tiff");
    ds.OnScanningFinished += OnScanFinished;
    ds.AcquireAsync(collector, true); // scanning has started
}

private void OnScanFinished(object sender, DataSource.ScanningFinishedEventArgs e){
    e.Colletor.Dispose(); // scanning has finished
}
```

Collectors

Collectors can be used to collect scanned images (see chapter: Collectors). They are either provided by the user to the acquire method, or the method will create the `ImageCollector` and return it to the user (depending on which overloaded function is called).

Transfer Mechanism

For acquiring, the user can specify one of four transfer mechanisms:

- Native Transfer (`TwSX.Native`): the data source delivers a whole image at once in the form of a single bitmap.
- Buffered Memory Transfer (`TwSX.Memory`): the data source delivers an image in a series of chunks of a bitmap. This mode will trigger the progress event.
- File Transfer (`TwSX.File`): the data source doesn't deliver an image to the library but saves an image to a file specified by the user. Collectors are not used in this scenario.
- Buffered Memory File Transfer (`TwSX.MemFile`): the data source delivers an image in a series of chunks of an image file. This mode will trigger the progress event, and collectors are not used in this scenario.

Passing the desired `TwSX` value to the desired acquire method instructs which transfer mechanism to use for scanning.

```
TwSX mech; // set desired transfer mechanism
var collector; // collector must be null
// synchronous scanning
ds.Acquire(collector, showUI, closeUIAfterAcquire, mech);
// asynchronous scanning
ds.AcquireAsync(collector, showUI, closeUIAfterAcquire, mech);
```

Other parameters

When acquiring user can specify some additional things:

- should scanner's GUI be displayed
- if the GUI is displayed should it automatically close when scanning is finished
- number of images to scan in the current batch
    - -1 instructs to acquire all available images
    - positive non-zero numbers limit the number of images to acquire

Events

During the acquiring process several events can be monitored:

- `OnSingleImageAcquired` is triggered when one image is acquired. The user can access this image in the form of `System.Drawing.Bitmap`.
- `OnBatchFinished` is triggered when all images from a single job are acquired. For example, all documents from feeder are scanned, or a single document from the flatbed is scanned.
- `OnScanningFinished` is triggered when scanning ends. For example, the user closed scanner's GUI.
- `OnMemoryTransferProgressUpdate` is triggered when a chunk of the image is delivered to this library. It is useful only for when `TwSX.Memory` or `TwSX.MemFile` transfers are used.
- `OnErrorEvent` triggered when something went wrong during acquiring process. For example: a paper jam, the device went offline.

Collectors

Collectors are objects which are used to collect acquired images. Collectors are typically instantiated by the user and passed to the data source through some of the acquiring methods. While it is the preferred way to deal with images, one is not obligated to use collectors. Collectors can be used for multiple scans; new images will be appended to the old ones.

Collectors can only be used when the transfer mechanism is Native or Buffered Memory (see chapter: Transfer Mechanism).

There are several types of collectors:

ImageCollector
Most commonly used collector and most powerful. Can save collected images to various formats: bitmap, jpeg, png, tiff, multipage tiff, pdf and multipage pdf. It uses the hard drive to temporary store scanned images. This feature can be disabled and images will be stored in memory, but then there is a risk of low memory exception.

ImageCollectorPdf
Stores all acquired images to the single PDF. It doesn't use a hard drive to store temporary images, and it can process the image while the next one is being scanned. The file will be saved on disposing.

ImageCollectorTiffMultipage
Stores all acquired images to the single tiff. It doesn't use a hard drive to store temporary images, and it can process the image while the next one is being scanned. The file will be saved on disposing.

ImageCollectorPdfSinglepage
Stores all acquired images to separate PDFs. It also doesn't use a hard drive to store temporarily.

ImageMultiCollector
Used when the functionality of the several different collectors should be combined; for example: when one must save images to pdf and tiff at the same time. Constructed with multiple collectors. Appends scanned images to every collector that it owns.

One can also create one’s own collector by implementing the IImageCollector interface.

IMPORTANT: All collectors must be disposed after use.

Scanning with different driver bitness

Scanning drivers are almost always 32-bit, but some scanners also provide 64-bit drivers. During development it's important to take the right approach depending on the bitness of the drivers and the application being developed.

32-bit drivers

Whenever possible, for scanning with 32-bit drivers only 32-bit applications should be preferred. For 64-bit applications a workaround exists in the form of `Bridgex86`.

32-bit applications

Scanning with 32-bit drivers in 32-bit applications yields the best performance. Scanning is performed in the same way as described so far.

64-bit applications with Bridgex86

Scanning with 32-bit drivers in 64-bit applications is made possible by using `Bridgex86` which provides a different workflow with dedicated methods for required scanner actions. This approach however has a slight performance impact and scanning can only be done synchronously. It's also not a complete feature set of everything that TwainScanning has to offer, but the most important features are available. Even though `Bridgex86` is a reliable alternative, whenever possible, we recommend using the regular workflow.

Each `Bridgex86` operation returns a `Result` object which holds the success `Status` of the operation, the result `Value` of the operation, and any `WarningMessages` or `ErrorMessages` that might have occurred. If for any operation the scanner is not specified, then the default scanner will be used.

Working with data sources (scanners)

The `Bridgex86.GetAllDevices()` method provides all scanners installed on the system. And the `Bridgex86.GetDefaultDevice()` method provides the default scanner.

```
var allScannersResult = Bridgex86.GetAllDevices(); // gets all installed scanners
if (allScannersResult.Status == StatusType.OK)
{
    Console.WriteLine("Found scanners:");
    foreach (var scanner in allScannersResult.Value) // scanners are in Value property
        Console.WriteLine(scanner);
}
else
    Console.WriteLine("No scanners found!");

var defaultScannerResult = Bridgex86.GetDefaultDevice(); // gets the default scanner
if (defaultScannerResult.Status == StatusType.OK)
    Console.WriteLine("Default scanner is: " + defaultScannerResult.Value); // Value
property can also be a single value
else
    Console.WriteLine("No default scanner found!");
```

Working with settings (capabilities)

A string list of all supported capabilities is provided by the `Bridgex86.GetAllSupportedCapabilities()` method. To check if a specific capability is supported use the `Bridgex86.GetIsSupportedCapability()` method, checks are not case sensitive. And to get all

values supported for a specific capability use the dedicated get method for that capability. For example, to get all supported color modes use the `Bridgex86.GetSupportedColorModes()` method. Due to how drivers provide values for supported capabilities, most values are returned as strings or booleans. A list of all current scanner settings is provided by using the `Bridgex86.GetCurrentDeviceSettings`() method, any unsupported settings will be null.

```
string scanner = "ScannerXYZ";
// if scanner is omitted then the default scanner is used
var allCapsResult = Bridgex86.GetAllSupportedCapabilities(scanner); // gets all
supported capabilities
Console.WriteLine("Supported capabilities:");
foreach (var cap in allCapsResult.Value) // all supported cap. in Value property
    Console.WriteLine(cap);

var pageSupportResult = Bridgex86.GetIsSupportedCapability("PageSize", scanner); //
checks support for specific capability by name (not case sensitive), alias can also be
used
bool isSupported = pageSupportResult.Value; // check Value to see if supported
if (isSupported)
    Console.WriteLine("It's supported");
else
    Console.WriteLine("It's not supported");

var colorsResult = Bridgex86.GetSupportedColorModes(scanner); // gets all supported
values for capability by using dedicated method
Console.WriteLine("Supported color modes:");
foreach (var color in colorsResult.Value) // all page sizes in Value property
    Console.WriteLine(color);

var currentResult = Bridgex86.GetCurrentDeviceSettings(scanner); // gets all current
settings
Console.WriteLine("Current scanner settings:");
Console.WriteLine("Scanner:" + currentResult.Value.Scanner); // scanner for which
settings were returned
Console.WriteLine("Color mode:" + currentResult.Value.ColorMode);
Console.WriteLine("Page size:" + currentResult.Value.PageSize);
// same approach for remaining settings
```

Acquiring (scanning)

The `Bridgex86.Acquire()` method requires an output file name and optionally a `ScanSettings` object with the settings required for scanning. Values for settings can be set explicitly or by selecting them from the list of supported values for that particular setting. For the settings which were not set, the default scanner values will be used. Any issues during scanning can be checked by inspecting the success `Status` and the `WarningMessages` and `ErrorMessages` properties.

```
string scanner = " ScannerXYZ ";
string outputFileName = @"C:\SomeFolder\Some\name.jpeg";

var settings = new ScanSettings(); // holds all scan settings, if a property (or the
entire object) is omitted then scanner defaults are used
settings.Device = scanner; // default scanner is used if omitted
settings.TransferMechanism = TwSX.Native;
settings.ShowUI = false; // show UI from driver software or not
settings.CloseUIAfterAcquire = true; // close UI from driver software or not
var pageCap = Bridgex86.GetSupportedPageSizes(scanner);
settings.PageSize = (TwSS)Enum.Parse(typeof(TwSS), pageCap.Value[0]); // some values
require parsing
settings.AutoFeed = true;
settings.ColorMode = TwPixelType.RGB;
settings.DuplexEnabled = true;
settings.ImageCount = -1; // scan all available images or limit number of images to
scan
settings.ImageQuality = 80;
settings.Resolution = new ScanResolution(300); // resolution can be individual for x
and y or common
settings.MultiPageScan = false; // scan into single file with multiple pages (tiff and
pdf only)

var scanResult = Bridgex86.Acquire(outputFileName, settings); // performs the scan

// check status and display related messages
if (scanResult.Status == StatusType.OK)
    Console.WriteLine("Scan successful!");
else if (scanResult.Status == StatusType.Warning)
    Console.WriteLine("Scan warnings:\n" + scanResult.WarningMessages);
else if (scanResult.Status == StatusType.Error)
    Console.WriteLine("Scan errors:\n" + scanResult.ErrorMessages);
else
    Console.WriteLine("Unknown issue during scan!");
```

64-bit drivers

32-bit applications

To use 64-bit drivers the application needs to be 64-bit, they can't be used in 32-bit applications.

64-bit applications

Despite the fact that 64-bit drivers should be usable in 64-bit applications, it is required to explicitly allow such scanning by setting the `GlobalConfig.Force64BitDriver` property. Scanning without setting it will fail with an exception being thrown. Scanning is performed in the same way as described so far. To check if the running process is 64-bit use the `GlobalConfig.Is64BitProcess` property.

```
if (GlobalConfig.Is64BitProcess) // checks if process is 64-bit
    GlobalConfig.Force64BitDriver = true; // forces using 64-bit drivers
// continue scanning as usual
using (var dsm = new DataSourceManager(this))                // opens TWAIN
using (var ds = dsm.OpenSource(dsm.SelectDefaultSourceDlg())) // opens default source
using (var collector = ds.Acquire(true))                     // acquires images
{
    collector.SaveAllToJpegs(@"C:\SomeFolder\Some\name.jpeg");// saves images to disk
}                                                            // closes ds and dsm
```

Appendix: List of capabilities

- AsyncDeviceEvents
  - DeviceEvent
- AudibleAlarms
  - Alarms
  - Volume
- AutomaticAdjustments
  - AutomaticSenseMedium
  - AutoDiscardBlankPages
  - AutomaticBorderDetection
  - AutomaticColorEnabled
  - AutomaticColorNonColorPixelType
  - AutomaticCropUsesFrame
  - AutomaticDeskew
  - AutomaticLengthDetection
  - AutomaticRotate
  - AutoSize
  - FlipRotation
  - ImageMerge
  - ImageMergeHeightThreshold
- AutomaticCapture
  - NumberOfImages
  - TimeBeforeFirstCapture
  - TimeBetweenCaptures
- AutomaticScanning
  - AutoScan
  - CameraEnabled
  - CameraOrder
  - CameraSide
  - ClearBuffers
  - MaxBatchBuffers
- BarCodeDetection
  - Enabled
  - SupportedTypes
  - MaxRetries
  - MaxSearchPriorities
  - SearchMode
  - SearchPriorities
  - Timeout
- Caps
  - ExtendedCaps
  - SupportedCaps
  - SupportedDats
- Color
  - ColorManagementEnabled
  - Filter
  - Gamma
  - IccProfile
  - PlanarChunky
- Compression
  - BitOrderCodes
  - CcittKFactor
  - Method
  - JpegPixelType
  - JpegQuality
  - JpegSubSampling
  - PixelFlavorCodes
  - TimeFill
- Device
  - Online
  - TimeDate
  - SerialNumber
  - MinimumHeight
  - MinimumWidth
  - ExposureTime
  - FlashUsed2
  - ImageFilter
  - LampState
  - LightPath
  - LightSource
  - NoiseFilter
  - OverScan
  - PhysicalHeight
  - PhysicalWidth
  - Unit
  - ZoomFactor
- DoublefeedDetection
  - Mode
  - DoubleFeedDetectionLength
  - DoubleFeedDetectionSensitivity
  - DoubleFeedDetectionResponse
- ImprinterEndorser

    - Endorser
    - Imprinter
    - ImprinterEnabled
    - ImprinterIndex
    - ImprinterMode
    - PrinterString
    - PrinterSuffix
    - PrinterVerticalOffset
    - PrinterCharRotation
    - PrinterFontStyle
    - PrinterIndexLeadChar
    - PrinterIndexMaxValue
    - PrinterIndexNumDigits
    - PrinterIndexStep
    - PrinterIndexTrigger
    - PrinterStringPreview
- ImageInformation
    - Author
    - Caption
    - TimeDate
    - ExtImageInfo
    - SupportedExtImageInfo
- ImageAcquire
    - ThumbnailsEnabled
    - AutoBright
    - Brightness
    - Contrast
    - Highlight
    - ImageDataSet
    - Mirror
    - Orientation
    - Rotation
    - Shadow
    - XScaling
    - YScaling
- ImageType
    - BitDepth
    - BitDepthReduction
    - BitOrder
    - CustHalftone
    - Halftones
    - PixelFlawor
    - PixelType
    - Threshold
- Language
    - TheLanguage
- MICR
    - Enabled
- Page
    - Segmented
    - Frames
    - MaxFrames
    - Sizes
- Duplex
    - Mode
    - Enabled
- Feeder
    - Autofeed
    - ClearPage
    - Alignment
    - Enabled
    - Loaded
    - Order
    - Pocket
    - Prepare
    - PaperDetectable
    - PaperHandling
    - ReacquireAllowed
    - RewindPage
    - Type
- PatchCodeDetection
    - Enabled
    - SupportedTypes
    - MaxSearchPriorities
    - SearchPriorities
    - SearchMode
    - MaxRetries
    - Timeout
- PowerMonitoring
    - BatteryMinutes
    - BatteryPercentage
    - PowerSaveTime
    - PowerSupply
- Resolution
    - X
    - Y
    - XNativeRes
    - YNativeRes

- Transfer
    - JobControl
    - ImageCount
    - Compression
    - ImageFileFormat
    - Tiles
    - UndefinedImageSize
    - Mechanism
- UserInterface
    - CameraPreviewUI
    - CustomDSData
    - CustomInterfaceGuid
    - EnableDSUIOnly
    - Indicators
    - IndicatorsMode
    - UIControllable

Appendix: Examples

Minimal example

Minimal but useful example.

```
// Minimal example using scanners UI
using (var dsm = new DataSourceManager(this))                // opens TWAIN
using (var ds = dsm.OpenSource(dsm.SelectDefaultSourceDlg())) // opens default source
using (var collector = ds.Acquire(true))                     // acquires images
{
    collector.SaveAllToJpegs(@"C:\SomeFolder\Some\name.jpeg");// saves images to disk
}                                                            // closes ds and dsm
```

Scanning in Windows Form

```
public partial class ScanningForm : Form
{
    DataSourceManager dsm = null;
    DataSource ds = null;
    public ScanningForm()
    {
        dsm = new DataSourceManager(this);
        InitializeComponent();
    }
    private void buttonScann_Click(object sender, EventArgs e)
    {
        dsm.SelectDefaultSourceDlg(); // sets default scanner
        ds = dsm.OpenSource();        // opens default scanner
        ds.AcquireAsync(onFinishedScanning, true, true, TwSX.Memory, -1);
    }
    private void onFinishedScanning(ImageCollector collector)
    {
        collector.SaveAllToMultipageTiff(@"C:\someFolder\someFile.tiff"); //saving
        collector.Dispose();   //disposing collector
        ds.Dispose();          //disposing scanner
        ds = null;
    }
    private void ScanningForm_FormClosing(object sender, FormClosingEventArgs e)
    {
        if (ds != null) ds.Dispose(); ds = null;
        if (dsm != null) dsm.Dispose(); dsm = null;
    }
}
```