EZTwain Pro 4.0 User Guide

A developer's guide to the EZTwain library version 4.0

Si, Quattro!
Cosa, quattro re? Quattro cilindri?
Quattro formaggio?

Hyperlinks	EZTwain Pro page: http://www.eztwain.com/eztwain4.htm Support: http://www.atalasoft.com/products/dotimage/forums
Author	Spike McLarty for Atalasoft
Revised	2/28/2011
Copyright	© Atalasoft, Inc. All rights reserved.
Trademarks	EZTwain Pro is a trademark of Atalasoft, Inc. Microsoft and Windows are trademarks of Microsoft Corporation. All other trademarks are the property of their respective owners.

Table of Contents

Introduction

This guide describes how to use the Atalasoft EZTwain library to add scanning or image-acquisition to a Microsoft® Windows® application. If you don't know much about TWAIN, the image-input standard that EZTwain is built on, don't worry – the necessary concepts and explanations are included.

For changes from previous versions of EZTwain, see Appendix 1 - History.

What is EZTwain? EZTwain is a Windows DLL that provides an easy-to-use wrapper for the TWAIN API. TWAIN is the most widely supported API for controlling scanners, and downloading images from cameras. EZTwain makes TWAIN easier for developers by radically reducing their learning and programming effort – which means fewer bugs, lower cost, more predictability, shorter schedules, and we believe fewer support problems.

With EZTwain, you can

- Acquire an image from a TWAIN-compliant device, bringing the image into memory or writing it immediately to a file, with one call.
- Select the output file format to be BMP, JPEG, PNG, DCX, TIFF, or PDF.
- Display the TWAIN dialog that allows the user to select among his or her TWAIN devices, or - enumerate the devices and present a list to the user, or - select a specific device by name.
- Suppress the normal user interface presented by a device, and take control of the scanning process from your program.
- Restrict or pre-select the scanning mode (B&W, Grayscale, RGB Color), the bit-depth, resolution, transparent versus reflective media, brightness, contrast, threshold, auto-brightness, duplex, and any other options offered by the device through TWAIN.
- Detect and control a document feeder (ADF).
- Scan multiple pages, discard blank pages, deskew (straighten) crooked pages.
- Collect scans into multi-page TIFF, DCX, or PDF files.
- Load, examine, display, and write image files in all supported formats.
- Query any property that your TWAIN device offers, and manipulate that property in any way allowed by TWAIN and the device.
- Upload images to a server via HTTP in any supported file format.

Atalasoft maintains, supports, and licenses EZTwain, as well as other tools for TWAIN developers. For more information, please visit www.atalasoft.com.

Overview

EZTwain Components

The EZTwain Pro Toolkit setup offers to install two sets of files:
Shared EZTwain DLLs
Developer Files
(or both)

If you choose to install the Shared DLLs, the DLLs listed in the table below are copied to the System folder:
C:\Windows\System32 (on most 32-bit versions of Windows)
C:\Windows\SysWOW64 (on 64-bit Windows)

If you choose to install Developer Files, the DLLs listed below are copied into
C:\Program Files\EZTwain\Redist (32-bit Windows)
C:\Program Files(x86)\EZTwain\Redist (64-bit Windows)

These are the DLLs that you use and distribute with your applications. If you are concerned about disk space or file sizes, the table below will help you decide which DLLs are required for your application.

See also: **How to Redistribute EZTwain with your Application**, page 12

Note that on 64-bit Windows, 32-bit processes that access the 'System32' folder are redirected to SysWOW64 – which means that if your 32-bit code loads e.g. "\Windows\System32\Eztwain3.dll", this will still work on 64-bit Windows.

Shared EZTwain DLLs

Eztwain3.dll	DLL containing the EZTwain Pro functions. It is *not* an ActiveX control or COM server, and does not need to be registered.
EZT4Jpeg.dll	*Optional* DLL. Required to read and write **JPEG**, **TIFF** or **PDF** format files.
EZT4Tiff.dll	*Optional* DLL to read and write TIFF files. *Requires EZT4Jpeg.dll.*
EZT4Pdf.dll	*Optional* DLL to write PDF files and to a limited extent read them. *Requires EZT4Jpeg.dll.*
EZT4Png.dll	*Optional* DLL to read and write PNG files
EZT4Gif.dll	*Optional* DLL to read and write GIF files.
EZT4Dcx.dll	*Optional* DLL to read and write DCX files.
EZT4Symbol.dll	*Optional* DLL to provide barcode recognition.
EZT4Curl.dll	*Optional* DLL to provide network file transfer (HTTP)
EZT4Ocr.dll	*Optional* DLL to interface to Transym OCR engine.

EZTwain Developer Files

When you run the EZTwain Pro Toolkit setup, if you choose to install Developer Files the toolkit setup will create a file structure under the Program Files folder:

\Program Files\EZTwain (32-bit Windows)
\Program Files(x86)\EZTwain (64-bit Windows)

The table below describes the contents of this file structure.

Developer Files

.	EZTwain install folder - Contains the Licensing Wizard, EZTwain license, this User Guide, Twirl,Twister, DosadiLog, EZTCheck, History.txt, Readme.txt.
.\Access	Contains eztwain.bas declaration file for use in VBA, and several small customer-donated Microsoft Access databases that use EZTwain.
.\Alpha5	eztwain. - declarations for Alpha Five.
.\BCB	Eztwain4.lib and Eztwain.h, for use in Borland C++ Builder programs.
.\Clarion	eztwain.clw and eztwain4.lib for Clarion
.\CSharp	eztwain.cs declaration file for C#
.\CSharp Sample	Sample C# program
.\dBASE	eztwain.h declarations for use with dBASE & co.
.\Delphi	EZTwain.pas declaration file and a small sample application for Borland Delphi 6.
.\Java	eztwain.java - Experimental JNA-based binding.
.\LabVIEW	Eztwain.h – for use with Import Shared Library
.\LotusScript	eztwain.lss declaration file for Lotus Notes/Domino.
.\Perl	Eztwain.pl declaration file, and a small sample program in Perl.
.\PowerBASIC	eztwain.inc declaration file.
.\PowerBuilder	Eztwain.txt - declarations and constant definitions to use EZTwain from Sybase PowerBuilder.
.\Progress	eztwain.i – file containing external declarations to use EZTwain from Progress 4GL.
.\Python	eztwain.py – experimental Python binding using ctypes
.\Redist	Copies of all the redistributable EZTwain DLLs.
.\Static	contains EZT4MT.LIB - a static link library version of EZTwain. See **How To: Statically Link to EZTwain Pro.**
.\VB	EZTwain.bas declaration file, and a small sample

	application for Visual Basic
.\VB.NET	EZTwain.vb declaration file for use in VB.NET
.\VB.NET Sample	Sample VB.NET program
.\VC	Microsoft Visual C++ files: EZTwain.h and EZTwain4.lib TWAIN.H – TWAIN API, in case somebody needs it.
.\VFP	Visual FoxPro declaration file.

How-To Guide

How To: Use the Code Wizard to get started

Our Code Wizard supports the following languages:

- Borland Delphi (5 thru 8)
- C# for .NET
- LotusScript
- Microsoft Visual C++ (6 or 7) with MFC
- Microsoft Visual C (6 or 7)
- PowerScript for PowerBuilder
- Visual Basic (5, 6 or 7) including VBA
- Visual FoxPro (7 & 8)
- VB.NET
- WinDev (English et Français)

For these languages, you should launch the Code Wizard (under Start - Programs - EZTwain) and step through it to generate code for some simple task like selecting the default TWAIN device, or doing a scan with default settings. The Wizard includes instructions for bringing EZTwain into your application. Then review the sections below to see if there are any specific comments for your language.

Microsoft Visual Basic 5, 6, or 7

Run the Code Wizard to get started - see above.

Sample: The \vb folder contains a small EZTwain sample application called VBTwerp, which has been tested with Visual Basic 5.

The EZTwain installer places the EZTwain DLLs in the System32 folder, so your program should not have trouble finding them.

See Also: Converting between DIBs and VB Pictures (p. 51)

C# and VB.NET

Run the Code Wizard to get started - see above.

In the EZTwain folder (usually under Program Files), there are sample programs in folders named: **VB.NET Sample Application** and **CSharp Sample Application**.

From .NET, EZTwain is basically just a big **friend** class with a lot of public/static functions. There are a few points to be aware of:

1. All of the TWAIN_xxx functions have had the TWAIN_ prefix stripped, so they are just EZTwain.xxx. For example, TWAIN_Acquire(0,0) becomes EZTwain.Acquire(0,0).
2. All of the constants defined in EZTwain.vb or EZTwain.cs need to be qualified just as the functions do e.g. EZTwain.TWPT_BW, EZTwain.EZT_TEXT_NORMAL, and so on.

3. Unfortunately, two functions, TWAIN_Set and TWAIN_Get, conflict with the VB.NET keywords 'Set' and 'Get', so they are aliased to EZTwain.SetCap and EZTwain.GetCap. You are not likely to need these, but just in case.
4. EZTwain works with quite a few kinds of 'platform' (native Windows API) handles, such as DIB handles, HBITMAPs and HWND Window handles. These are pretty much all translated to System.IntPtr per Microsoft's recommendation. As a result you must be careful to read the documentation to see exactly which kind of handle you are - er - handling. Don't mix them up!
5. We provide a function DIB_ToImage that copies an EZTwain DIB into a .NET Image object, for VB.NET. Contact us if you need the equivalent code for C#.

Please contact Atalasoft technical support if you encounter any problems using EZTwain Pro from .NET. We are committed to resolving such problems promptly.

Microsoft Visual FoxPro

Run the Code Wizard to get started - see above.

For a nice introduction to using EZTwain from VFP, see this article by Mike Lewis of Mike Lewis Consultants Ltd:
http://www.ml-consult.co.uk/foxst-29.htm

See the general discussion under Redistributing EZTwain with an Application.

Borland Delphi

Run the Code Wizard to get started - see above.

There is a very small Delphi EZTwain sample application, created with Delphi 6, in the EZTwain\Delphi folder, including the project file: You should just be able to double-click the project file to open the sample in Delphi.

Caution: The sample converts images from the DIB format delivered by EZTwain, into the TBitmap format favored by Delphi. We **do not** recommend converting images from DIB to TBitmap *and back* because information (particularly DPI) can be lost if you do this.

Microsoft Visual C++

Run the Code Wizard to get started - see above. Note: It is possible to statically link EZTwain Pro, see How To: Statically Link to EZTwain, p 11.

LotusScript

PowerScript - PowerBuilder

Run the Code Wizard to get started - see above. Note that we provide one declaration file named 'eztwain.txt' for PowerBuilder version 10 and later, and another named 'eztwain-pb9.txt' for PowerBuilder 9 and earlier, due to incompatible declaration syntax. Make sure you use the right one.

How To: Use EZTwain from other languages

LabVIEW

We do not offer any pre-written LabVIEW code, but several customers have used EZTwain from LabVIEW.

For links and the most up-to-date information about using EZTwain from LabVIEW, visit our LabVIEW Support Page at: http://www.eztwain.com/ezt3labview.htm

Perl

We do not understand Perl! But one of our customers helped us create a Perl declaration file and a small sample. They can be found in the EZTwain toolkit folder \Program Files\EZTwain\Perl

Borland C++ Builder (BCB)

EZTwain Pro has been used successfully from BCB 6.0 – The toolkit includes a header file EZTwain.h, and a link-library Eztwain4.lib, which by default is copied to this folder: \Program Files\EZTwain\BCB

Warning: The default handling of floating-point exceptions is different in Microsoft languages, and Borland languages. Certain TWAIN device drivers will generate fatal run-time exceptions if they are invoked from Borland applications unless precautions are taken. See our advisory: http://www.eztwain.com/borland-issue.htm

Microsoft Access (VBA)

We don't claim any expertise with Microsoft Access, but several customers who are regular Access users have donated sample code for *accessing* (cough) EZTwain Pro. Their databases can be found in: \Program Files\EZTwain\Access

Clarion, dBASE (dBASE+, VDB), PowerBASIC, Progress 4GL

If you look in the toolkit folder \Program Files\EZTwain you will find sub-folders for these languages, and perhaps others. In those subfolders will be a file containing external declarations for all EZTwain functions. There are also blocks of constant definitions, which you may use at your convenience, and in some cases there are small sample programs.

Java

We do not provide a Java binding, but customers have suggested JNI, JNA, and JNative all as useful. Search our Forum for more details:
http://www.eztwain.com/Forums/

Other languages

For other development platforms, please contact Atalasoft support via our Forums
http://www.atalasoft.com/products/dotimage/forums?forumid=16

How To: Statically Link to EZTwain

This section assumes you are using Microsoft Visual C/C++. For other languages or compilers, you are responsible for adapting the following advice.

When you install the EZTwain Pro toolkit, you designate a main folder for the various developer files. Under that folder is a subfolder called Static containing a statically linkable library: EZT4MT.LIB. This library contains almost all (exceptions noted below) of the functions from Eztwain DLL.

Note: The static library *directly* supports only BMP file format: To read or write any other file format, the same DLLs as described in Shared EZTwain DLLs must be present and loadable.

- Use the function declarations from eztwain.h, and link to EZT4MT.LIB.
- EZT4MT.LIB links to LIBCMT or LIBCMTD, the multithreading version of the C runtime library. EZTwain uses multithreading internally, so your application must also be compiled for multithreading.
- Call EZTWAIN_Attach() before calling any other EZTwain function. Make a matching call to EZTWAIN_Detach() before terminating the application.
- TWAIN_ViewFile and DIB_View are missing - they depend on dialog resources that are not included in the .LIB

How To: Redistribute EZTwain with your Application

There are two main questions when redistributing EZTwain: Where to put the DLLs, and how to obtain a license (key).

Where to Put the DLLs

If you are preparing a software package to be distributed with EZTwain, you have three main alternatives:

Alternative 1. Follow the lead of the EZTwain Developer Kit and install the EZTwain DLLs in the System (System32) folder. The sample apps and various definition files are set up for this, using unqualified references to "eztwain4.dll".

If you have your own installer, it *must* compare versions before overwriting the EZ*.DLL files in the system folder, and only overwrite a higher version with a lower after strenuous warnings to the user. This is a specific obligation under the EZTwain Pro License Agreement. Any responsible install tool will do this by default, or at least offer it as an option.

This alternative leaves you exposed to the following risk: Another product could be installed after yours, replacing your EZTwain DLL's with higher-versioned ones. *Or* vice-versa: Your application could replace older DLLs installed by a previous application. Either way, if the new DLLs are not sufficiently backward compatible, one *or both* of the applications involved can stop working. We make every effort to keep our DLLs backwards-compatible, but it cannot be guaranteed. This is one form of "DLL Hell" and is a risk with using any DLL installed in a System folder.

Alternative 2. If your application compiles to an EXE file, you can install the EZTwain DLLs in the same folder as the .EXE. Use an unqualified DLL reference as in Alternative 1. Under this alternative, your application will always (we believe) load and use those specific DLLs.

Alternative 3. For dynamic-binding languages like VB and FoxPro, you can use fully qualified paths for the EZTwain DLL – If you decide to do this, replace all occurrences of "Eztwain4.DLL" in the definition file with the full path of the DLL .e.g. "c:\Program Files\Eztwain\Redist\Eztwain4.dll" Under this alternative, your installer must install the EZTwain DLLs in the same specific folder on every target machine, or must have a way at run-time to find and load the DLLs.

How To: Obtain a License Key

Redistribution of the EZTwain DLLs is only allowed by the EZTwain Pro License if you purchase a Universal Redistribution License key. For this key, you provide a *vendor name*, and add a call to TWAIN_SetVendorKey in your software. See details under TWAIN_SetVendorKey in the Function Reference section below. For details and ordering information, run the Licensing Wizard, or browse to:

http://www.eztwain.com

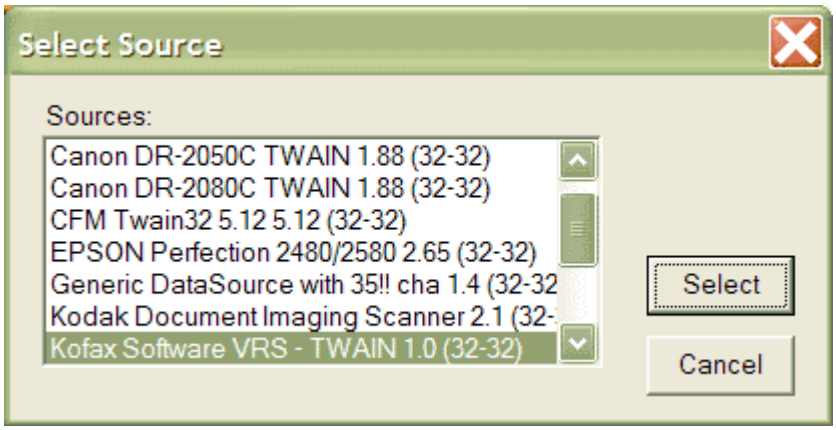
How To: Select a Device for Input

Displaying the Select Source Dialog

You can implement the Select Source command with one EZTwain call:

```
TWAIN_SelectImageSource(0);
```

This function displays the TWAIN Select Source dialog, with a list of all the installed TWAIN Sources on the system:



The user can select the new *default TWAIN device* or, they can cancel. If they OK this dialog, TWAIN remembers the new default device.

Note that the Select Source dialog lists Sources (TWAIN drivers), not physical devices: It will list devices even if they are currently off-line or unplugged. Also, two devices that use the same driver will only appear *once* in the TWAIN device list.

Please don't make the user go through the Select Source dialog each time they want to acquire!

Enumerating the available sources

If you would like to display your own list of TWAIN Sources, or find out the exact name of a Source, you can use TWAIN_GetSourceList and TWAIN_GetNextSourceName. The Code Wizard, installed as part of the EZTwain Pro toolkit, will generate the code to do this in variety of languages.

Opening a Source by Name

If you would like to acquire from a specific device, you will need to know its exact name. You can enumerate the names of the installed devices - see above. Then you can open a specific source this way:

```
if (TWAIN_OpenSource("Logitech Camera") == 1) {
      TWAIN_AcquireToFilename(0, "frame.bmp");
}
```

This tries to open the named device, and if successful, acquires an image and stores it in a bmp file. All of the Acquire functions work this way – If a source is open, they use it, and otherwise they open and use the default source.

How To: Acquire an Image

To acquire a single image from the default TWAIN device, using the device's user interface, and store it in a BMP file, call

```
TWAIN_AcquireToFilename(0, "filename.bmp")
```

This makes an excellent test of:

1. Your ability to invoke EZTwain,
2. That EZTwain DLL (Eztwain4.dll) is where your program can find it,
3. TWAIN is correctly installed on the computer, and
4. The default TWAIN device being correctly installed and operational.

A word about the 'default TWAIN device' - If there is only one TWAIN device installed in the system, then that device is the default TWAIN device. Otherwise, it is the last TWAIN device selected by the user in the Select Source dialog.

To acquire an image into memory:

```
hdib = TWAIN_Acquire(0);
if (hdib != 0) {
      DIB_WriteToFilename(hdib, "last_scan.tif");
      DIB_Free(hdib); hdib = 0;
}
```

This acquires a single image from the default device, formats it in memory as a DIB, and returns a handle to it - A handle, not a pointer. Then it writes the DIB out as a TIFF file, and frees the DIB.

Note that the returned object is a DIB. There is a Windows object commonly called a Bitmap, short for Device-Dependent Bitmap or DDB – a DIB is *not* a DDB! A DIB is a completely different animal.

If something goes wrong with the transfer, the return value will be NULL (0).

The DIB_Free call is needed to release the memory holding the image. If you don't do that, the image sits around taking up memory until your program exits.

How To: Negotiate Scanning Parameters

TWAIN requires, reasonably enough, that you must have a Source open before you can ask it about its settings and properties, called *capabilities* in TWAIN-speak.

TWAIN also imposes the restriction that settings can only be set while the Source is open and before it has been enabled. 'Enabled' in TWAIN means given the go-ahead to acquire images.

To cut to the chase, here's a C fragment for scanning a 1-bit, 300dpi, B&W image:

```
if (TWAIN_OpenDefaultSource()) {
      // DS is now in State 4 (Open)
      TWAIN_SetPixelType(TWPT_BW);
      TWAIN_SetBitDepth(1);
      // (probably redundant for BW)
      TWAIN_SetUnits(TWUN_INCHES);
      TWAIN_SetResolution(300.0);
      hdib = TWAIN_Acquire(0);
}
```

This code does no error checking on the Set functions, even though some of them will certainly fail on some devices. For example, almost any webcam will reject a pixel type of TWPT_BW, and probably a resolution setting of 300 – it might reject *any* attempt to set resolution.

It is recommended to select a pixel type first, then to set the bit depth – some devices maintain bit depths for each pixel type.

Setting the units to inches is a precaution that introduces interesting issues. In theory, resolution is defined as *samples per unit of measure*. So when we set a resolution of 300, we are setting 300 dpi only if the current unit of measure is inches (dpi = dots per inch, right?) If we are in a metric country – and there's only one country that isn't metric – then the DS might be configured to use centimeters, and 300 would mean 300 samples per centimeter. In practice, TWAIN specifies the default unit of measure as inches, so almost all Sources open with their units set to inches. **However** –some webcams and video capture devices open with units set to pixels! Technically non-compliant, these devices are likely to reject any attempt to set their resolution anyway.

How To: Scan a Multipage Document

Multipage files are image files that can hold multiple pages or sequential images. The multipage file formats supported by EZTwain are TIFF, PDF, and DCX. DCX is rarely used so we will not mention again.

To scan from the default TWAIN device into a multipage TIFF file takes one call:

```
TWAIN_AcquireMultipageFile(hwnd, "multipage.tif");
```

The first parameter is the window handle of your main window – if you can't easily obtain this, just pass a 0 - EZTwain will use the handle of the active window if any, or it will create an invisible window if there is no active window. The second parameter is the filename to create - EZTwain uses the extension to select the format of the file: .TIF, .TIFF, or .MPT means write a TIFF file, .PDF means write a PDF, etc.

TWAIN_AcquireMultipageFile will present the scanner's user interface, and will accept scans from the scanner (or images from a camera or webcam) until the user closes the scan dialog. A few devices will close their window automatically after sending one image - in some cases EZTwain can detect this and will prompt the user asking if there are more images. All of this is invisible to your application.

If you do not want to display the device's user interface, you can use code like the following. If the default TWAIN device is a scanner with an *Automatic Document Feeder* (ADF), this code will scan all the pages in the feeder:

```
if (TWAIN_OpenDefaultSource()) {
    TWAIN_SetHideUI(1);          // ask for no user interface
    TWAIN_SetResolution(300);    // ask for 300 DPI
    TWAIN_SetPixelType(TWPT_BW); // black & white
    TWAIN_SelectFeeder(1);       // from feeder (if possible)
    TWAIN_EnableDuplex(1);       // both sides (if supported)
    TWAIN_AcquireMultipageFile(hwnd, "multipage.tif");
}
if (TWAIN_LastErrorCode() != 0) {
    TWAIN_ReportLastError("Error during scanning.");
}
```

If the current TWAIN device is a flatbed scanner, this code will immediately scan from the flatbed, then display a small message box asking if there are more pages to scan. As long as the user answers 'Yes', the TWAIN_AcquireMultipageFile function will continue to scan another page.

It's important to specify at least the resolution and pixel type when the scanner user interface is suppressed, otherwise you have no way of knowing what settings the scanner will use.

How To: Hide the Source User Interface

In theory

Make the following call before any Acquire calls, and images will arrive from your device with no distracting dialogs or windows on screen:

```
TWAIN_SetHideUI(1)
```

In practice

Some devices will refuse to cooperate and will display their user interface (UI) anyway. A few particularly bad TWAIN device drivers will crash when used this way.

Even in No-UI mode, many devices will display a status or progress box while scanning or transferring data. Sometimes the progress box can be suppressed [see TWAIN_SetIndicators, p. 129.]

If the device is not connected and powered up, if the paper jams, or if anything else happens that requires human attention, most devices will display an error dialog. Suppressing such error messages requires low-level Windows programming outside the scope of TWAIN or EZTwain.

You can count on almost all desktop scanners, departmental scanners, and high-volume scanners to scan well without showing their UI.

Some webcams will transfer with no UI, for example the Logitech QuickCams.

You *cannot* count on a digital still camera (DSC) to transfer all its images from memory in No-UI mode. Last time we checked (2005) most DSCs ignore the request to hide their UI.

If you must automate a device that insists on displaying its user interface (almost any webcam, for example) you can try the function TWAIN_AutoClickButton.

How To: Control a Document Feeder (ADF)

Here's a sample of code to drive a scanner with an automatic document feeder, suppressing the Source user interface:

```
if (!TWAIN_OpenDefaultSource()) {
        return;                         // unable to open device?
}
TWAIN_SetHideUI(1);                     // hide the user interface
TWAIN_SelectFeeder(1);                  // tell device: 'use feeder'
TWAIN_SetAutoScan(1);                   // tell device: 'scan ahead'
TWAIN_EnableDuplex(1);                  // tell device: 'both sides, if you can'
// OK - start scanning
do {
        HANDLE hdib = TWAIN_Acquire (0);
        if (hdib==0) {
                // something went wrong…
                break;
        }
        ProcessThisPage(hdib);
        DIB_Free(hdib);
} while (!TWAIN_IsDone());
TWAIN_CloseSource();
```

Be aware that TWAIN_SelectFeeder can fail on a scanner that looks for all the world to have a document feeder. One of the biggest players in consumer scanners has shipped a variety of sheet-feeding models that deny (through TWAIN) having feeders. The above code fragment will probably drive such devices, even though both the Feeder-related functions fail.

TWAIN_SetAutoFeed and TWAIN_SetAutoScan can fail, this is harmless. Scanners that support this feature will scan at full speed when these are set, and scanners that don't will still feed pages when Acquire requests them.

How To: Skip Blank Pages

If you are using TWAIN_AcquireMultipageFile, you can skip blank pages by simply calling

```
TWAIN_SetBlankPageMode(1)
```

This tells AcquireMultipageFile to detect and discard blank pages. Use TWAIN_SetBlankPageThreshold to adjust the threshold for 'blankness'.

If you are not using AcquireMultipageFile, you will need to call functions to detect blank pages and handle them in your control flow. Below we have taken a standard multipage scanning loop in Visual Basic generated by the Code Wizard, and added code to ignore blank pages:

```
Dim fileName As String
Dim hdib As Long
fileName = "c:\mydoc.pdf"           ' Output to PDF
Call TWAIN_SetHideUI(0)             ' Hide scanner dialog
If TWAIN_OpenDefaultSource()=1 Then
   Call TWAIN_SelectFeeder(1)       ' Pull from ADF
   Call TWAIN_SetXferCount(-1)      ' All available pages
   Call TWAIN_SetAutoScan(1)        ' Scan ahead if you can
   Call TWAIN_SetMultiTransfer(1)   ' Allow multi acquire
   Call TWAIN_BeginMultipageFile(fileName)
   Do
      hdib = TWAIN_Acquire(Me.hwnd) ' Get next image
      If hdib=0 Then
         Exit Do
      End If
      ' We decide that < 2% 'ink' coverage means blank.
      ' Only write non-blank pages to file:
      If Not DIB_IsBlank(hdib, 0.02) Then
         Call TWAIN_DibWritePage(hdib)
      End If
      ' Always remember to free the image from memory:
      Call DIB_Free(hdib)
   Loop While TWAIN_State()>=5
   Call TWAIN_CloseSource()
   Call TWAIN_EndMultipageFile()
End If
If TWAIN_LastErrorCode()<>0 Then
   Call TWAIN_ReportLastError("Scan error.")
End If
```

How To: Read Patch Codes

We'll need to enable patch code detection, by setting the TWAIN capability ICAP_PATCHCODEDETECTIONENABLED to TRUE. With the scanner open in EZTwain, this would be something like:

```
TWAIN_SetCapability(ICAP_PATCHCODEDETECTIONENABLED, 1)
```

or EZTwain.SetCapability(ICAP_PATCHCODEDETECTIONENABLED, 1)

Then we need to tell EZTwain to ask for 'extended image info' after each scan, specifically the patch code. Something like this:

```
TWAIN_EnableExtendedInfo(TWEI_PATCHCODE, 1)
```
That 1 may need to be TRUE, True or true, depending on programming language.

After each image is received by using TWAIN_Acquire or similar single-image scanning function, you can read the patch code that was found, if any, with code similar to this:

```
If TWAIN_ExtendedInfoItemCount(TWEI_PATCHCODE) > 0 Then
   one or more patch codes found - assume only 1 for now!
   patch_code = TWAIN_ExtendedInfoInt(TWEI_PATCHCODE, 0)
```

According to the TWAIN standard, the returned patch codes are:

TWPCH_PATCH1 1
TWPCH_PATCH2 2
TWPCH_PATCH3 3
TWPCH_PATCH4 4
TWPCH_PATCH6 5 *genius...*
TWPCH_PATCHT 6

How To: Append to PDF, TIFF & DCX Files

When the function TWAIN_SetFileAppendFlag is called with a non-zero argument, it tells EZTwain to append to existing PDF, TIFF and DCX files, instead of overwriting their contents. Example:

```
TWAIN_SetFileAppendFlag(1);
TWAIN_AcquireMultipageFile(0, "papertrail.pdf");
```

This will scan pages from the default TWAIN device, appending them to the file "papertrail.pdf" until the operator closes the device dialog. The effect of adding the call to TWAIN_SetFileAppendFlag is that if papertrail.pdf *already exists* the new pages are added to it. If papertrail.pdf does not exist, it is created.

The File Append Flag applies to TWAIN_AcquireMultipageFile as above, and to any other EZTwain function writing a PDF, TIFF or DCX file.

Obscure Exception: This cannot be used with TWAIN_AcquireFile, because with that function the TWAIN driver writes the file, not EZTwain.

How To: Check for Device On-Line

You can't! TWAIN added a capability for this rather late in the game, and did a bad job: Very few TWAIN devices handle this correctly.

Summary: There is no known general way to check if a TWAIN device is on-line.

How To: Do Other Random Stuff

Note: Your syntax may vary. In some languages, TWAIN_*function* is called as EZTwain.*function*, and all other functions and constants may need to be prefixed with "EZTwain." Some EZTwain Pro functions accept or return *boolean* values: Depending on your language these may be 0 and 1, **true** and **false**, TRUE and FALSE, etc.

To turn off the 'auto rotation' feature of certain scanners, such as the Canon network scanners:

```
TWAIN_SetCapBool(ICAP_AUTOMATICROTATION, 0)
```

Function Reference

Functions – Application Name & Licensing

Note: If you installed the EZTwain Pro developer kit on your system, the terms of the EZTwain License are available under Start – Programs – EZTwain, in License.txt in the EZTwain folder, and on the web at:
http://www.eztwain.com/EZTwain_Pro_License.pdf

See How To: Obtain a License Key, page 12

TWAIN_SetAppTitle

```
void TWAIN_SetAppTitle(string Title)
```

Sets the *application title*, which is used several ways:

- As the title (caption) of any message boxes displayed by EZTwain.
- By a few Sources, in their progress box e.g. "Transferring image to <app title>"
- In theory, it could be used by a Source to react specially to an application.

If you call TWAIN_ApplicationLicense, you do not need to call TWAIN_SetAppTitle. If you do not set the application title at all, the application is given a tacky default title such as "Application using EZTwain".

TWAIN_ApplicationLicense

```
void TWAIN_ApplicationLicense(string pzAppTitle, int nAppKey)
```

Sets the title of the application (as far as EZTwain is concerned) and unlocks EZTwain using a Single Application Redistribution Key. It unlocks EZTwain Pro, if the numeric key (nAppKey) matches the application title. Make sure you use the exact string used to purchase the license – it is listed with the key in the grant-of-license e-mail. This should be the first EZTwain call your application makes, other than TWAIN_LogFile.

TWAIN_SetApplicationKey

```
void TWAIN_SetApplicationKey(int nAppKey)
```

Similar to TWAIN_ApplicationLicense above, but called after TWAIN_SetAppTitle or TWAIN_RegisterApp. It unlocks EZTwain using a Single Application Redistribution Key, if the key matches the application title you have set.

TWAIN_SetVendorKey

```
void TWAIN_SetVendorKey(string pzVendorName, int nKey)
```

Unlocks EZTwain using a *Universal Redistribution License* key – Sometimes also called a Vendor License Key. Make sure you use the exact string that was entered

for 'Vendor' on the order – it is listed with the key in the grant-of-license e-mail. This should be the first EZTwain call your application makes, except for TWAIN_LogFile.
Use TWAIN_SetAppTitle to tell EZTwain the name of the application.

TWAIN_OrganizationLicense

```
void TWAIN_OrganizationLicense(string pzOrganization, int nKey)
```

Unlocks EZTwain using an *In-House Application License* key – Sometimes also called an Organization License key. Make sure you use the exact string that was entered for 'Organization' on the order – it is listed with the key in the grant-of-license e-mail. This should be the first EZTwain call your application makes, except for TWAIN_LogFile.

Use TWAIN_SetAppTitle to tell EZTwain the name of the application.

TWAIN_SingleMachineLicense

```
BOOL TWAIN_SingleMachineLicense(string Prompt)
```

This function is for use in applications that will deployed on a small number of computers, using the EZTwain Pro Single Machine License – which is a per-machine license. Call this function when your application starts up, perhaps even during installation of your application, and it will prompt as needed for a license key on each machine.

Previously a Single Machine License could only be installed on a computer by first installing the entire EZTwain Pro Developer Toolkit, then running the EZTwain Pro Licensing Wizard. This function replaces that procedure, allowing the developer to build the licensing process into his or her application.

When called, this function checks to see if EZTwain Pro is licensed to run on this computer at this time.

1. If the running copy of EZTwain is licensed *in any way* - including a trial license - this function silently returns TRUE(1) to the caller.

2. If no license is found, this function displays a 'license needed' dialog, with the Prompt argument at the top of the dialog. If an application title has been set with TWAIN_SetAppTitle, it is used as the title of the dialog.

   The user is told how to obtain a license key and how to enter it. Once a key is entered and found valid, it is stored on the computer and the function returns TRUE(1). **Note:** For this function to accept and store a license key, it must be running with Administrative privileges.

3. If the user cancels the licensing dialog, this function will normally return FALSE(0), which means that EZTwain is not licensed to run on the computer.

You can use the Prompt argument to direct users to an in-house support person who can help resolve licensing problems. For example:

```
if (!TWAIN_SingleMachineLicense("Contact Willy Codewell x8000!")) {
    exit(255);
}
```

If the Prompt string is empty, a default prompt similar to the following is used:

> "This application (<apptitle>) uses EZTwain Pro, which requires a valid numeric License Key."

Functions – Image Acquisition

General Comments

EZTwain is built on a principal of brevity: You should only need to make calls or pass parameters, where EZTwain cannot reasonably guess what you want.

Advice: When developing, we strongly suggest that you start with a simple call to TWAIN_AcquireToFilename (if you need a file), TWAIN_AcquireMultipageFile (to scan a multipage document) or TWAIN_Acquire (if you need an image in memory.) Then add additional calls *one by one* to get the exact behavior you ultimately want.

By default, all Acquire functions shut down TWAIN before returning. If you are doing a series of transfers from the same device, it is much faster to leave the device open between transfers. You can use a multipage scanning function to do this, or use TWAIN_SetMultiTransfer (p 34) to keep the device open for multiple transfers.

All Acquire functions load TWAIN and then open and enable the default Source, *if needed.* EZTwain goes to great lengths to track the TWAIN State, and will generally move TWAIN automatically from state to state as needed. For example, if you start by calling TWAIN_OpenSource, EZTwain makes the necessary calls to transition from State 1 to State 4. If you then call an Acquire function, EZTwain sees that there is a Source open, and proceeds with acquisition from that device.

By default, the Acquire functions tell the device to display its user interface. Use TWAIN_SetHideUI to change this.

Single Image Scanning Functions

TWAIN_AcquireToFilename

```
int TWAIN_AcquireToFilename(HWND hwndApp, string pszFile)
```

Acquire an image and save it to a file. If the filename contains a standard extension (.bmp, .jpg, .jpeg, .tif, .tiff, .png, .pdf, .gif, .dcx) then the file is saved in the implied format. Otherwise the file is saved in the default save format– see TWAIN_SetSaveFormat (p 94).

If pszFile is NULL or an empty string, the user is prompted for the file name *and format* with a standard Save File dialog. Only available and appropriate formats are presented in the Save File dialog. In this case if you need to know the filename the user chose, you can call TWAIN_LastOutputFile (p).

See also TWAIN_Acquire below.

Return values:

0	success.
-1	the Acquire failed.
-2	file open error (invalid path or name, or access denied)
-3	invalid DIB, or image incompatible with file format, or...
-4	writing failed, possibly output device is full.
-10	user cancelled File Save dialog

The minimal use of EZTwain is to call this function with null arguments:

```
ErrCode = TWAIN_AcquireToFilename(0, "")
```

TWAIN_Acquire

```
HANDLE TWAIN_Acquire(HWND hwndApp)
```

Acquires a single image, from the currently selected Source, using EZTwain's preferred transfer mode.

The return value is a handle to global memory containing a DIB, a Device-Independent Bitmap. There are numerous functions to examine, modify, and save these DIB images. Remember to call DIB_Free on each DIB when you are done with it!

To acquire an RGB image with 8 bits/channel, you need to do something like this:

```
TWAIN_SetPixelType(TWPT_RGB);
hdib = TWAIN_Acquire(hwnd)
if (hdib) {
      // process image
      DIB_Free(hdib);
}
```

Multi-image Scanning Functions

TWAIN_AcquireMultipageFile

```
int TWAIN_AcquireMultipageFile(HWND hwndApp, string Filename)
```

Acquire multiple images into a single output file.

If the filename ends with a recognized extension, the file is written in the implied format: .TIF/.TIFF/.MPT => TIFF, .DCX => DCX format, and .PDF => PDF.

If the filename has no recognized extension, the file is written in the *default multipage format* as set by TWAIN_SetMultipageFormat (p 35).

If Filename is NULL or the empty string, the user will be prompted for the file name. The only format offered will be the current default multipage format. If you use this feature, you can call TWAIN_LastOutputFile to obtain the actual filename.

Return values:

0	success
-1	the Acquire failed.
-2	file open error (invalid path or name, or access denied)
-3	invalid DIB
-4	writing failed, possibly output device is full.
-10	user cancelled File Save dialog

Other functions that affect this function:

TWAIN_SetHideUI	hide or show the scanner's user interface.
TWAIN_SetAutoDeskew	automatically deskew each page.
TWAIN_SetBlankPageMode	discard blank pages.
TWAIN_SetMultiTransfer	leave device open when the function returns.

If TWAIN_SetHideUI is 0 [the default case] then the device UI is shown, and AcquireMultipageFile will transfer images until the user closes the device dialog.

If SetHideUI is 1, then the device UI is hidden and AcquireMultipageFile will transfer images until the device indicates that it has no more images available (technically, until it goes to State 5). Exception: In the case of a device that does not have a feeder, AcquireMultipageFile will prompt the user after each page, asking if there are more pages to scan: See TWAIN_PromptToContinue.

TWAIN_BlankDiscardCount returns the number of blank pages discarded by TWAIN_AcquireMultipageFile.

TWAIN_MultipageCount can be called during or after a multipage acquire: It returns the number of images written to the most recently created multipage file. See also TWAIN_AcquireCount just below.

If you want to set scanning parameters (resolution, pixeltype...) first open the source (see OpenDefaultSource or OpenSource) then negotiate the settings using the capability functions, and then call AcquireMultipageFile.

TWAIN_AcquireToArray

```
int TWAIN_AcquireToArray(HWND hwnd, HDIB ahdib[], int nMax)
```

Scan and store images in an array.
Very similar to TWAIN_AcquireMultipageFile, see that function for more details.
A return value of N >= 0 means N images were scanned and stored without error.
(N=0 if the job ended without error after 0 pages.)
Any unused entries in the array are set to 0 (NULL)
In case of error, returns a negative value and any scanned images are discarded.

TWAIN_AcquireImagesToFiles

```
int TWAIN_AcquireImagesToFiles(HWND hwndApp, string Filename)
```

Similar to TWAIN_AcquireMultipageFile above, but writes each image to a separate file.

If the filename is NULL or points to the empty string, the user is prompted for the name of the first file using a standard Save dialog.

As with TWAIN_AcquireToFilename, if the filename contains a standard extension (.bmp, .jpg, .jpeg, .tif, .tiff, .png, .pdf, .gif, .dcx) then the file is saved in the implied format. Otherwise the file is saved in the default save format– see TWAIN_SetSaveFormat (p 94).

Auto-numbering

The first image acquired is written to the specified filename. Subsequent filenames are *auto-numbered* according to this algorithm:

1. If the previous filename, excluding extension, does not end in one or more digits, then it is treated as if it ended with '0'.
2. If the previous name, excluding extension, ends in a sequence of d digits specifying the number n, then the next filename is created by replacing the sequence of digits with a sequence of digits representing the number n+1, padded with leading 0's to make it at least d digits long.

Examples

Filename (1st File)	2nd File	3rd File
Document.tif	Document1.tif	Document2.tif
Page98.jpg	Page99.jpg	Page100.jpg
Invoice00001.pdf	Invoice00002.pdf	Invoice00003.pdf

Return values from TWAIN_AcquireImagesToFiles

≥0	The number of files written. This could be 0 if the scanner dialog is displayed and the user closes the dialog without any scans, or if blank pages are being discarded and all of the scanned pages are (classified as) blank.
<0	An error. The codes are the same as TWAIN_AcquireMultipageFile.

For more detailed error information, use TWAIN_ReportLastError, TWAIN_LastErrorCode, etc.

See also TWAIN_AcquireCount and TWAIN_BlankDiscardCount.

TWAIN_AcquirePagesToFiles

```
int TWAIN_AcquirePagesToFiles(HWND hwnd, int nPPF, string sFile)
```

Like AcquireImagesToFiles, but can handle duplex scanning and multipage files. See Controlling Duplex Mode.

hwnd = parent window. Use 0 (NULL) if you can't obtain the window handle.

nPPF = *physical pages per file*.
If the scanner is scanning single-sided (simplex) then each file will receive nPPF images. If the scanner is scanning duplex, the number of images written into each file will be 2 x nPPF. The result is that each file represents nPPF pieces of paper, whether you are scanning duplex, or simplex.

sFile = name of first file.
We recommend including the extension to specify the format.
If the filename is NULL or points to the empty string, the user is prompted for the name of the first file.
Files are *auto-numbered*, see TWAIN_AcquireImagesToFiles above.

Return value: Same as TWAIN_AcquireImagesToFiles.

Example: Assume you want to load a batch of 2-page single-sided forms into your ADF-equipped scanner and turn them into individual PDF files. Following the code needed to open and configure the scanner for paper size, pixel type, resolution, duplex(off) and so on, the actual scan can be performed by:

```
TWAIN_AcquirePagesToFiles(hwnd, 2, "form0001.pdf")
```

This will scan 2 pages at a time until the feeder runs empty, writing the first two pages into form0001.pdf, the next two pages into form0002.pdf, and so on.

If you later need to scan forms that are printed on both sides - and assuming your scanner can scan duplex - you would insert a line to select duplex mode before starting the scan:

```
TWAIN_EnableDuplex(TRUE)
TWAIN_AcquirePagesToFiles(hwnd, 2, "form0001.pdf")
```

The AcquirePages function adapts correctly, scanning 2 pages at a time, which now produces 4 images (front, back, front, back), and collecting each 2-page form into a sequentially numbered PDF file, as before.

If you condense your original 2-page single-sided forms onto 1 double-sided page, then you have to change the number of pages per file:

```
TWAIN_EnableDuplex(TRUE)
TWAIN_AcquirePagesToFiles(hwnd, 1, "form0001.pdf")
```

This takes each 1-page form in the feeder, scans both sides, and writes a corresponding PDF file containing the front and back image.

TWAIN_AcquireCount

```
int TWAIN_AcquireCount()
```

Returns the number of images received from the scanner during the most recent multipage Acquire function (such as TWAIN_AcquireMultipageFile).

TWAIN_BlankDiscardCount

```
int TWAIN_BlankDiscardCount()
```

Returns the number of pages discarded as blank during the most recent multipage Acquire function. See TWAIN_SetBlankPageMode (p 36).

TWAIN_PromptToContinue

```
BOOL TWAIN_PromptToContinue(HWND hwnd)
```

Prompt the user asking if they want to continue scanning.
Return TRUE(1) if user responds affirmatively, FALSE(0) if not.

If the parameter is a valid Windows window-handle, that window is used as the parent of the prompt message box, otherwise the foreground window of the current task/process is used.

If you have called TWAIN_SetScanAnotherPagePrompt with a (non-empty) string, that string is used as the prompt message.

Otherwise, a standard prompt is used:
The prompt is automatically translated based on thread locale, which defaults to application locale, which defaults to user locale, which defaults to system locale.

Languages: Danish, Dutch, English, French, German, Italian,
Norwegian, Polish, Portuguese, Spanish, Swedish.

TWAIN_SetScanAnotherPagePrompt

```
TWAIN_SetScanAnotherPagePrompt(string pzPrompt)
```

Sets the prompt message for the "Scan another page?" prompt.
See TWAIN_PromptToContinue above. This prompt is also used by all the multipage Acquire functions in certain circumstances.

TWAIN_AcquireFile

```
int TWAIN_AcquireFile(HWND hwndApp, int nFF, string pszFile)
```

Acquire one image and write it to a file using TWAIN File Transfer Mode. This is an exotic transfer mode, not supported by all TWAIN devices. Do not use this function unless you have a specific reason and understand the consequences. If you just want to scan one image or page to a file, use **TWAIN_AcquireToFilename**.

Warning: File Transfer Mode is *not* supported by all TWAIN devices, and when it is supported, often BMP is the only supported file format.

You can open a Source and then use TWAIN_SupportsFileXfer to see if the DS supports File Transfer Mode.

You can use TWAIN_Get(ICAP_IMAGEFILEFORMAT) to get an enumeration of the available file formats, and CONTAINER_ContainsValue to check for a particular format you are interested in. See Appendix 2 - Working with Containers, p 161.

nFF can be any file format supported by the DS, see the TWFF_* constants in twain.h for the list of allowed formats. A compliant DS should at least support TWFF_BMP, but as usual there are no guarantees.

If pszFile is NULL or an empty string, the user is prompted for the file name in a standard Save File dialog. If you use this feature, you can call TWAIN_LastOutputFile to obtain the actual filename.

Return values (Note, this is not an error code like AcquireToFilename!)
TRUE(1) for success
FALSE(0) for failure
Use GetResultCode/GetConditionCode for details.
If the user cancels the Save File dialog, the result code will be TWRC_CANCEL

Functions – Global Modes & Queries

TWAIN_EasyVersion

```
int TWAIN_EasyVersion()
```

Returns the version number of EZTwain Eztwain4.dll, multiplied by 100.
So e.g. 416 as a return value means EZTwain Version 4.16

TWAIN_IsAvailable

```
int TWAIN_IsAvailable()
```

Call this function any time to find out if TWAIN is installed on the system. It takes a little time on the first call, after that it's extremely fast. It returns 1 if the TWAIN Source Manager is installed and can be loaded, 0 otherwise.

TWAIN_SetHideUI / TWAIN_GetHideUI

```
TWAIN_SetHideUI(int fHide)
int TWAIN_GetHideUI()
```

These functions control the 'hide source user interface' flag. This flag is initially FALSE(0), but if you set it non-zero, then when a source is enabled it will be asked to hide its user interface. *Note this is a request - some sources will ignore it.*
See: How To: Hide the Datasource User Interface.
If the user interface is hidden, you will probably want to set at least some of the basic acquisition parameters yourself – see Negotiating Scanning Parameters . See also: HasControllableUI

TWAIN_SetMultiTransfer / TWAIN_GetMultiTransfer

```
TWAIN_SetMultiTransfer(int fYes)
int TWAIN_GetMultiTransfer()
```

These functions query and set the 'multiple transfers' flag.
By default, the multi-transfer flag is FALSE(0). This means that EZTwain closes down TWAIN after each image transfer (TWAIN_AcquireXXX).

If the multi-transfer flag is non-zero: After an Acquire, the Source is left open and enabled to allow additional images to be acquired in the same session. The programmer may need to close the Source after all images have been transferred, for example by calling TWAIN_CloseSource or TWAIN_UnloadSourceManager

See How To: Transfer Multiple Images

TWAIN_DisableParent / TWAIN_GetDisableParent

```
void TWAIN_DisableParent(int fYes)
int TWAIN_GetDisableParent (void)
```

Disable the parent window during all TWAIN_Acquire functions.
(The parent window is the window you pass to the Acquire function. Typically this is your main application window or dialog.) By default this setting is TRUE - the parent window, if you pass it in, is disabled during an Acquire.

Note 1: If you set this to FALSE, your window can receive user input while an Acquire is in progress, and your code must be prepared for this.
Note 2: Some TWAIN Sources will disable the parent window on their own, and EZTwain cannot prevent this.

TWAIN_SetMultipageFormat
TWAIN_GetMultipageFormat

```
int TWAIN_SetMultipageFormat(int nFF)
int TWAIN_GetMultipageFormat()
```

Select/query the default multipage file save format, the file format used to write multipage files *when the file format cannot be inferred from the file extension.*

If you use a recognized extension in the name of your multipage file - such as .tif, .tiff, .mpt, .pdf or .dcx, then the file will be written in the implied format. The file extension overrides SetMultipageFormat.

The default when EZTwain is loaded is MULTIPAGE_TIFF.

Multipage format values:

0	MULTIPAGE_TIFF
1	MULTIPAGE_PDF
2	MULTIPAGE_DCX

SetMultipageFormat returns:

0	success,
-1	invalid/unrecognized format (bad parameter value)
-3	format is currently unavailable (missing/bad DLL)
-7	Multipage support is not installed.

Functions – Post-Processing

Post-processing is the industry term for everything you do to an image after it has been scanned or captured, before you store it or pass it on.

TWAIN_SetAutoCrop/TWAIN_GetAutoCrop

```
TWAIN_SetAutoCrop(int nMode)
int TWAIN_GetAutoCrop()
```

Select the *auto-crop* mode.
Auto-crop mode attempts to crop off very dark areas on the outside of each incoming image during scanning. The available auto-crop modes are:

0 no auto crop.
1 auto crop using software algorithm

TWAIN_SetAutoContrast/TWAIN_GetAutoContrast

```
TWAIN_SetAutoContrast(int nMode)
int TWAIN_GetAutoContrast()
```

Select the *auto-contrast* mode.
Auto-contrast mode attempts to automatically improve the contrast of each incoming image during scanning. It works best on text documents. See DIB_AutoContrast for more details. The available auto-contrast modes are:

0 no auto contrast.
1 auto contrast using software algorithm

TWAIN_SetAutoDeskew/TWAIN_GetAutoDeskew

```
TWAIN_SetAutoDeskew(int nMode)
int TWAIN_GetAutoDeskew()
```

Select the 'auto-deskew' mode.
Auto-deskew attempts to straighten up scans that are slightly crooked, up to about ±10 degrees. The available auto-deskew modes are:

0 no auto deskew.
1 enable scanner deskew (ICAP_AUTOMATICDESKEW) if supported; if not supported by scanner, deskew in software.

TWAIN_SetBlankPageMode / TWAIN_GetBlankPageMode

```
TWAIN_SetBlankPageMode(int nMode)
int TWAIN_GetBlankPageMode()
```

Set or get the *Skip Blank Pages* mode. When this mode is 1, blank pages are automatically discarded by TWAIN_AcquireMultipageFile.
When this mode is 0 (the default), EZTwain does not look for blank pages or treat them in any special way.
See TWAIN_SetBlankPageThreshold (below) for more details.

TWAIN_SetBlankPageThreshold / TWAIN_GetBlankPageThreshold

```
TWAIN_SetBlankPageThreshold(double dDarkness)
double TWAIN_GetBlankPageThreshold()
```

Set or get the blank page threshold, which affects the operation of *Skip Blank Pages* mode. The blank page threshold is a number between 0 and 1.0. If the number of 'dark' pixels on a page divided by the total number of pixels on the page is less than this threshold, the page is considered 'blank'.

The initial blank page threshold is: 0.02 (= 2% dark pixels).

See DIB_IsBlank (p 64) and DIB_Darkness (p 64) for more details about the algorithm for computing 'dark' pixels.

TWAIN_SetAutoOCR / TWAIN_GetAutoOCR

TWAIN_SetAutoOCR(int nMode)
int TWAIN_GetAutoOCR()

Sets or gets the auto-OCR mode. By default this mode is ***OFF(0)***. When this mode is on(1), EZTwain applies OCR, if available, to each incoming scanned page or image and temporarily stores the result. Also in this mode, if you scan directly to PDF format using TWAIN_AcquireToFilename or TWAIN_AcquireMultipageFile, the OCR'd text is written invisibly to each PDF page, to facilitate indexing and searching. If you are scanning individual pages you can call OCR_Text or OCR_GetText to retrieve the text found on the last scanned page.

The currently selected OCR engine is used: See OCR_SelectEngine and related functions. Caution: If OCR fails for some reason in auto-OCR mode, an error is recorded (see TWAIN_LastErrorCode, TWAIN_ReportLastError) but the scanning function may report success.

TWAIN_SetAutoNegate/TWAIN_GetAutoNegate

```
TWAIN_SetAutoNegate(BOOL bYes)
BOOL TWAIN_GetAutoNegate()
```

Controls the 'auto-negate' mode. Unlike most post-processing modes, this mode is ***on*** by default.
Auto-negate mode analyzes each incoming B&W (1-bit/pixel) image during scanning. If the image is more than 80% black, it is 'negated' - black and white are reversed. A surprising number of scanners incorrectly deliver negative images under certain circumstances, and this mode compensates for that.

The available auto-negate modes are:

0	no auto negate.
1	auto negate (default)

Source (Device/Driver) Selection

TWAIN_SelectImageSource

```
int TWAIN_SelectImageSource(HWND hwnd)
```

This function posts the Source Manager's Select Source dialog box, which allows the user to see and possibly change the *default TWAIN device for their computer*. Note: If only one TWAIN device is installed on a system, it is selected automatically, so there is no need for the user to do Select Source. You should not require your users to do Select Source before Acquire.

Note that this dialog is displayed by the TWAIN Source Manager, not by EZTwain, and we have no direct control over its size, location, layout, etc. If you want a different UI, you can create your own using the functions below.

It usually works well to pass a 0 to this function: EZTwain then finds and refers to the 'Foreground Window', and the Select Source dialog acts as a modal child of that window. If for some reason that isn't what you want, you'll need to pass another valid HWND (Windows window-handle) to this function.

Returns TRUE(1) if the user OK'd a selection, FALSE(0) if:
a) The user cancelled the dialog
b) The Source Manager found no Sources installed
c) There was a failure before the Select Source dialog could be posted

To enumerate the available Sources, see the following two functions.

TWAIN_GetSourceList

```
int TWAIN_GetSourceList(void)
```

Fetches the list of sources into memory, so they can be returned one by one by TWAIN_GetNextSourceName, below.
Returns TRUE (1) if successful (at least one Source was found), FALSE (0) otherwise.

TWAIN_GetNextSourceName/TWAIN_NextSourceName

```
int TWAIN_GetNextSourceName(LPSTR pzName)
string TWAIN_NextSourceName()
```

TWAIN_GetNextSourceName copies the next source name in the list into pzName – up to 32 characters (ANSI) plus a terminating NUL (0 byte).
Returns TRUE (1) if successful, FALSE (0) if there are no more.

TWAIN_NextSourceName returns the next source name as a string. This function is not available in some languages. If there is no next source name, this function returns the empty string.

TWAIN_GetDefaultSourceName
TWAIN_DefaultSourceName

```
int TWAIN_GetDefaultSourceName(LPSTR pzName)
string TWAIN_DefaultSourceName()
```

TWAIN defines the 'default source' as a per-user or system-wide TWAIN setting: It is the default TWAIN device, similar to the Windows default printer. Like the default printer, most applications that have an Acquire or Scan command will use the current default source. However, some applications maintain their own private source device, independent of the TWAIN default source.

Until TWAIN 2.2, the default source could *only* be selected by the user, using the TWAIN Select Source dialog (see TWAIN_SelectImageSource.)

TWAIN_GetDefaultSourceName copies the name of the TWAIN default source into pzName. Up to 32 ANSI characters *plus* a terminating NUL (0 byte) are returned. Normally returns TRUE (1) but will return FALSE (0) if:

- the TWAIN Source Manager cannot be loaded & initialized or
- there is no current default source (e.g. no sources are installed)

DefaultSourceName returns the name of the TWAIN default source as a string. This function is not available in certain languages. If there is no default TWAIN source (see above) this function returns the empty string.

TWAIN_SourceName

```
char* TWAIN_SourceName(void)
```

Returns the name of the currently or last opened source, as a string. C/C++ developers: Note that this is always an 8-bit ASCII string.

See TWAIN_DefaultSourceName, to obtain the name of the default TWAIN device.

TWAIN_GetSourceName

```
void TWAIN_GetSourceName(LPSTR pzName)
```

Like TWAIN_SourceName, but copies the name string into its argument. Please allocate enough space: 64 char at least.

Functions – Extended Image Information

Extended Image Information was introduced in TWAIN 1.7 – It provides a way for a scanner to offer advanced image processing services, and send the results to a receptive application with each scanned image. For example, some higher-end scanners can detect barcode symbols or automatically deskew pages. Using this feature, an application can find out what barcodes were found on each incoming page, and the amount of skew that was corrected.

The EZTwain model of this feature is fairly simple. As part of configuring for a scan, the application uses TWAIN_EnableExtendedInfo to specify which type of extended information should be collected. Each type of information is represented by a numeric constant defined in the TWAIN standard, such as TWEI_BARCODETEXT or TWEI_SKEWORIGINALANGLE. After an image is received, using TWAIN_Acquire for example, we provide functions to count and read the collected information.

We have included the necessary TWEI_ constants in our declaration files, but for explanation and details you should contact us (www.eztwain.com) or visit the TWAIN website (www.twain.org) to obtain a copy of the TWAIN Standard.

TWAIN_IsExtendedInfoSupported

```
BOOL TWAIN_IsExtendedInfoSupported()
```

Asks the currently open device if it can generate Extended Image Info.
Returns TRUE(1) if yes, FALSE(0) if not. This will fail and record an error if no TWAIN device is currently open through EZTwain.

TWAIN_EnableExtendedInfo

```
BOOL TWAIN_EnableExtendedInfo(long eiCode, BOOL enabled)
```

Enable or disable collection of the specified kind of extended image info.
Each type of information is represented by an integer constant with prefix TWEI_ - see the EZTwain declaration file for your language. This function returns TRUE.

TWAIN_IsExtendedInfoEnable

```
BOOL TWAIN_IsExtendedInfoEnabled(long eiCode)
```

Return TRUE(1) if the specified extended image info is enabled.

TWAIN_DisableExtendedInfo

```
void TWAIN_DisableExtendedInfo()
```

Disables all extended image info – calling this function stops collection of extended image information.

Reading Extended Information

You can think of the extended image information as returning an array of 0 or more values for each enabled TWEI_ code. TWAIN_ExtendedInfoItemCount tells you how many values were returned for a given TWEI_ code. When you ask for a value, you always have to specify both the TWEI_ code and a value index. The value index is overkill in almost all cases: You will commonly specify an index of 0, meaning the first available value.

Remember: All of these functions refer to information collected from the last scan.

TWAIN_ExtendedInfoItemCount

```
long TWAIN_ExtendedInfoItemCount(long tweiCode)
```

The number of values available of the given info (TWEI_) type.

TWAIN_ExtendedInfoItemType

```
long TWAIN_ExtendedInfoItemType(long tweiCode)
```

Returns a number indicating the type of data returned for the specified extended info, using the same TWTY_ codes as CONTAINER_ItemType (see page 162).

TWAIN_ExtendedInfoInt

```
long TWAIN_ExtendedInfoInt(long tweiCode, long n)
```

Returns the integer value of the 'nth' item of the specified extended info.

TWAIN_ExtendedInfoFloat

```
double TWAIN_ExtendedInfoFloat(long tweiCode, long n)
```

Returns the (floating point) value of the 'nth' item of the specified extended info.

TWAIN_GetExtendedInfoString

```
BOOL TWAIN_GetExtendedInfoString(long tweiCode, long n, LPSTR
Buffer, long Bufsize)
```

Read the string value of the nth item of the specified info into Buffer, which has been allocated by the caller to hold Bufsize characters.
Note that the value returned is ASCII (byte) text, not unicode, and *always* includes an ending 0 byte, even if it must be truncated to fit.
Returns TRUE if the data was retrieved and could fit in the buffer, FALSE otherwise.

In all languages, the caller must ensure that the 3rd parameter (Buffer) has been allocated as a block of characters (bytes) and that the address of the first byte of the allocated buffer is passed to the function. In classic VB for example, this requires

passing the first buffer element by reference. Contact Technical Support if you have questions about this.

TWAIN_ExtendedInfoString

```
string TWAIN_ExtendedInfoString(long tweiCode, long n)
```

As above, but the string is returned as a temporary pointer to a 0-terminated ASCII string. In case of any failure, returns the empty string ("").

If your programming language has strings that are natively UNICODE, this function, if available at all, will return a native string and the comments above about ASCII/byte text do not apply: The string is converted to native format when it is returned from the function.

TWAIN_GetExtendedInfoFrame

```
BOOL TWAIN_GetExtendedInfoFrame(
      long tweiCode, long n,
      double *L, double *T, double *R, double *B)
```

Fetch the TW_FRAME value of the 'nth' item of the specified extended info. A *frame* is a TWAIN concept usually used to represent a rectangle – left, top, right, bottom. This is rarely used, but is here for completeness.

Functions – DIBs & Image Processing

Creating and Freeing DIBs

DIB_Allocate

```
HANDLE DIB_Allocate(int nDepth, int nWidth, int nHeight)
```

Create a DIB with the given dimensions. Resolution is set to 0. A default grayscale table is provided if depth <= 8. The image data is uninitialized i.e. garbage.

DIB_Create

```
HANDLE DIB_Create(int nType, int nWidth, int nHeight, int nDepth)
```

Create a DIB with the given pixel type, dimensions, and depth. See *Pixel Types*, page 44. If a depth of 0 is given, the default depth for the given pixel type is used. Resolution is set to 0. If the pixel type calls for a color table (TWPT_BW, TWPT_GRAY, or TWPT_PALETTE) a default color table is provided. The image data is uninitialized.

DIB_Copy

```
HANDLE DIB_Copy(HANDLE hdib)
```

Create and return a byte-for-byte copy of a DIB.

DIB_Free

```
void DIB_Free(HANDLE hdib)
```

Release the storage of the DIB.
Note: If hdib is NULL (0), it does nothing.

DIB_FreeArray

```
void DIB_FreeArray(??, int n)
```

Calls DIB_Free on the first n entries of a DIB handle array.

Querying DIB Properties

DIB_Width

```
int DIB_Width (HANDLE hdib)
```

Width of DIB, in pixels (columns)

DIB_Height

```
int DIB_Height (HANDLE hdib)
```

Height of DIB, in lines (rows)

DIB_PixelType

```
int DIB_PixelType(HANDLE hdib)
```

Returns a Pixel Type code that describes the format of the DIB's pixels.

EZTwain Pixel Types

Symbol	Code	Description
TWPT_BW	0	1-bit per pixel, black and white
TWPT_GRAY	1	grayscale, normally 8 but can be 4- or 16-bit
TWPT_RGB	2	RGB color, 24-bit (can also be 48-bit, and rarely 15, 16, or 32-bit)
TWPT_PALETTE	3	indexed color (image has a color table) 8 or 4-bit.
TWPT_CMY	4	CMY color, 24-bit
TWPT_CMYK	5	CMYK color, 32-bit

DIB_Depth / DIB_BitsPerPixel

```
int DIB_Depth (HANDLE hdib)
int DIB_BitsPerPixel (HANDLE hdib)
```

Number of bits per pixel.

DIB_SamplesPerPixel

```
int DIB_SamplesPerPixel(HANDLE hdib)
```

Number of samples (components or color channels) in each pixel.
B&W and gray pixels have 1 sample, RGB and CMY have 3.
CMYK has 4. Palette-color images are treated as having 3 channels.

DIB_BitsPerSample

```
int DIB_BitsPerSample(HANDLE hdib)
```

Number of bits per sample (channel, component) in each pixel.

For B&W and grayscale images, this is the same as the bits per pixel, because those formats have one sample per pixel.

For palette images, this will be 8, because the color values in a palette image are stored with 8 bits each for R, G, and B.

For RGB, CMY, and CMYK images, this function returns the number of bits used to represent each color channel or component - almost always 8, but EZTwain does have a limited ability to handle images that are 5-bit and 16-bit per channel.

DIB_XResolution / DIB_YResolution

```
double DIB_XResolution(HANDLE hdib)
double DIB_YResolution (HANDLE hdib)
```

Horizontal (x) or vertical (y) resolution of DIB in DPI (dots per inch)

DIB_PhysicalWidth / DIB_PhysicalHeight

```
double DIB_PhysicalWidth(HDIB hdib, int nUnits)
double DIB_PhysicalHeight(HDIB hdib, int nUnits)
```

Return the width(height), in the specified units, of the given image, calculated using its pixel width(height) and X(Y) resolution. If the resolution is 0, these return 0.

nUnits is one of the TWUN_ values - see page 122 - 0=inches, 1=cm, etc.

DIB_IsCompressed

```
BOOL DIB_IsCompressed(HDIB hdib)
```

TRUE(1) if the DIB's image data is compressed in memory, FALSE(0) otherwise. Compressed DIBs are only produced by an operation where you specifically request or enable creation of compressed DIBs.

DIB_Compression

```
int DIB_Compression(HANDLE hdib)
```

Returns a code specifying the type of compression used on a DIB's image data.
Uses the same codes as TWAIN_SetCompression.
TWCP_NONE means 'no compression' – the common case.

DIB_RowBytes

```
size_t DIB_RowBytes(HANDLE hdib)
```

Number of bytes needed to store one row of the DIB.

DIB_Size

```
int DIB_Size(HANDLE hdib)
```

The number of bytes of memory occupied by the DIB – header plus image data.

DIB_ColorCount

```
int DIB_ColorCount(HANDLE hdib)
```

Number of colors in color table of DIB.

DIB_ColorTableR / DIB_ColorTableG / DIB_ColorTableB

```
int DIB_ColorTableR(HANDLE hdib, int i)
int DIB_ColorTableG(HANDLE hdib, int i)
int DIB_ColorTableB(HANDLE hdib, int i)
```

Return the R,G, or B component of the ith color table entry of a DIB.
If i < 0 or >= DIB_ColorCount(hdib), returns 0.

Setting DIB Properties

DIB_SetResolution/DIB_SetResolutionInt

```
void DIB_SetResolution(HANDLE hdib, double xdpi, double ydpi)
void DIB_SetResolutionInt(HANDLE hdib, int xdpi, int ydpi)
```

Sets the horizontal or vertical resolution of the DIB. The 'Int' form is for languages that cannot easily pass double (64-bit floating point) parameters.

DIB_SetGrayColorTable

```
void DIB_SetGrayColorTable(HANDLE hdib)
```

Fill the DIB's color table with a gray ramp - so color 0 is black, and the last color (largest pixel value) is white. No effect if depth > 8. The DIB must already have a color table allocated.

DIB_SetColorTableRGB

```
void DIB_SetColorTableRGB(HANDLE hdib, int i, int R, int G, int
B)
```

Set the ith entry in the DIB's color table to the specified color. R G and B range from 0 to 255.

Reading and Writing DIB Data

DIB_ReadRow
DIB_ReadRowRGB
DIB_ReadRowGray
DIB_ReadRowChannel

```
void DIB_ReadRow(HANDLE hdib, int r, BYTE* prow)
void DIB_ReadRowRGB(HANDLE hdib, int r, BYTE* prow)
void DIB_ReadRowGray(HANDLE hdib, int r, BYTE *prow)
void DIB_ReadRowChannel(HANDLE hdib, int r, BYTE *prow, int c)
```

Read row r of the given DIB into buffer at prow.
Row 0 is the *top* row of the image, as it would be displayed.

DIB_ReadRow reads the *raw row data* from the DIB, including BGR pixels from 24-bit DIBs, 1-bit, 4-bit or 8-bit, 16-bit, or even 48-bit pixels.
DIB_ReadRowRGB converts each pixel into the nearest equivalent 3-byte RGB pixel.
DIB_ReadRowGray converts every pixel to an 8-bit grayscale or "brightness" value.
DIB_ReadRowChannel extracts the 8-bit channel or component of each pixel, as described in *Component Codes*, page 64

The caller is responsible for making sure there is enough room in the buffer (pointed to or referenced by the *prow* parameter.) Buffer sizes required are as follows:

Function	Bytes of buffer per row
DIB_ReadRow	DIB_RowBytes(hdib)
DIB_ReadRowRGB	3*DIB_Width(hdib)
DIB_ReadRowGray	DIB_Width(hdib)
DIB_ReadRowChannel	DIB_Width(hdib)

DIB_ReadData

```
void DIB_ReadData(HANDLE hdib, BYTE* pdata, int nbMax)
```

Read the entire DIB, including header, into a buffer. The 2nd parameter is the address (pointer to) the buffer. The 3rd parameter is the maximum number of bytes to read – which is usually the size of the buffer in bytes.

The number of bytes needed to hold the entire DIB is returned by DIB_Size.

DIB_WriteRow

```
void DIB_WriteRow(HANDLE hdib, int r, const BYTE* pdata)
```

Write data from buffer into row r of the given DIB.
Caller is responsible for ensuring that the buffer and row exist, etc.

DIB_WriteRowChannel

```
void DIB_WriteRowChannel(HANDLE hdib, int r, const BYTE* pdata,
int nChannel)
```

Write data from buffer into one color channel of row r of the given image.
When writing a 24-bit RGB image, valid channels are: 1=Red, 2=Green, 3=Blue.
When writing to a 32-bit RGBA image, channel 4=alpha.
When writing to an 8-bit gray image, channel 0 = gray.
When writing to a 24-bit CMY or 32-bit CMYK image, 1=Cyan, 2=Magenta, 3=Yellow, 4=blacK.
This function *should not be used* on any other image format.

Drawing (Rendering) DIBs

DIB_DrawOnWindow

```
void DIB_DrawOnWindow(HANDLE hdib, HWND hwnd)
```

Draws the DIB on the window.

The image is scaled to just fit inside the (client area of the) window, while keeping the correct aspect ratio. Any part of the window not covered by the image is left untouched, so will normally be filled with the window's background color.

DIB_DrawToDC

```
void DIB_DrawToDC( HANDLE hdib,    // DIB handle
                   HDC hDC,        // destination device context
                   int dx, int dy,// destination (x,y)
                   int w, int h,  // width and height
                   int sx, int sy // source (x,y) in DIB
                   )
```

Draws the DIB on the device context. Before using this call, you should have some understanding of the Windows GDI and Device Contexts.

Converting between DIB and other image formats

Microsoft, never content to offer one solution when it can offer two (or more), has over the years created at least five widely-used classes of image object:

1. The *Device-Independent Bitmap or DIB*.

   EZTwain Pro uses this format internally, representing each image as a global handle to a block of memory containing a DIB header immediately followed by the pixel data. DIBs can store a wide variety of image formats, and retain resolution (DPI) information. In native Windows API programming, the DIB is the standard general-purpose image format, although it is usually referenced using pointers rather than a global handle.

2. The *Device-Dependent Bitmap, DDB, or HBITMAP*.

   Often simply called a 'bitmap', and referenced in the Windows API by a handle called an HBITMAP, a DDB has a device-dependent pixel format (although actually the 1-bit format is standardized) and can only be manipulated by the video device driver that created it. DDBs do not store resolution (DPI) information. It is not generally meaningful to save a DDB to a file.

   Today, DDBs are only useful in two contexts: They are still the easiest way to embed fixed graphics as *resources* into a Windows program in C/C++. And if you need the absolute maximum speed, they are probably the fastest way to move pixels to and from the display.

   EZTwain can convert from DDB (HBITMAP) to DIB with DIB_FromBitmap.

3. The *DibSection*

   This is a strange hybrid object, an HBITMAP that wraps a DIB. Many languages and imaging classes (such as GDI+, .NET Image, Delphi TBitmap) do not easily accept DIBs but readily accept a DIBSection as an HBITMAP.

   EZTwain can convert a DIB to a DibSection with DIB_ToDibSection. Use this function when you need an HBITMAP. ***Note:*** DIB_ToDibSection frees the DIB.

4. The *Picture/IPicture/OLE Picture* object.

   This is a COM object wrapper for an image - it can wrap either a bitmap or a Windows Metafile (WMF). VB supports this format as the *Picture* type - when calling EZTwain from VB, you can use DIB_ToPicture and DIB_FromPicture.

5. The *Image* class

   The .NET framework introduced the Image class. When using EZTwain from VB.NET, you can use the function DIB_ToImage.

DIB_ToDibSection

```
HBITMAP DIB_ToDibSection(HANDLE hdib)
```

Convert the given DIB into a kind of bitmap called a *DIBSection*, which is a special kind of Windows native bitmap. The returned HBITMAP (bitmap handle) can be used with many Windows functions and controls. Many class libraries (such as .NET Image and Delphi TBitmap) also prefer this kind of bitmap.

Note: The input DIB is freed and can no longer be accessed or used.

DIB_FromBitmap

```
HANDLE DIB_FromBitmap(HBITMAP hbm, HDC hdc)
```

Reverse of DIB_ToDibSection: Converts a device-dependent bitmap into a DIB (Device-Independent Bitmap). If successful, the input bitmap is deleted.

Most programmers can pass 0 (NULL) for the HDC argument. The HDC (handle to device context) should only be used if you understand GDI programming and know that the HDC is compatible with the HBITMAP and contains useful color palette information. If the incoming HBITMAP is a DIBSection (for example from DIB_ToDibSection) the HDC is never needed and can always be 0.

DIB_ToImage/DibToImage

```
Function DIB_ToImage(ByVal hdib As System.IntPtr) As Image
Function DibToImage(ByVal hdib As System.IntPtr) As Image
```

Converts a DIB into a .NET Image object and returns it. Note that with this function, the input DIB is ***not*** freed - your code must free the DIB when you are done with it.

Converting between DIBs and VB Pictures

Visual Basic has a type called variously a Picture, StdPicture, or IPicture – which can be used alone, or in conjunction with the PictureBox control, whose *Picture* property holds a Picture object. EZTwain prefers to work with and store images as DIBs, so the following functions allow conversion between DIBs and VB Pictures.

Warning: VB's Picture objects do not retain *resolution* (DPI) – or we have not found the secret that makes it work. The resolution is what determines the implied physical (printed) size of an image, so if you need this information, you will have to somehow restore it when you convert from Picture to DIB. See p. 46 – DIB_SetResolution. We recommend not converting Pictures back to DIBs.

Note: The Picture type is replaced in .NET by the Image class, so these functions are not available in .NET languages - See DIB_ToImage.

The EZTwain Developer Kit contains a non-trivial sample application called VBTwerp written for VB 5.0, which demonstrates use of these and many other EZTwain functions.

DIB_ToPicture

```
Picture DIB_ToPicture(HANDLE hdib)
```

Reformat the given DIB into a Picture. **Note: This frees the input DIB – it can no longer be accessed or used.**

A Picture object can be assigned to the *Picture* property of a PictureBox on a form, causing the picture to be displayed. Discussion of the PictureBox control is beyond the scope of this document – see the VB documentation, and many Web resources. If you have a form *frmMain* containing a PictureBox named *ScannedPic*, this statement will scan an image and display it:

```
Dim hdib As Long
hdib = TWAIN_Acquire(frmMain.hwnd)
If hdib <> 0 Then
    Set ScannedPic.Picture = DIB_ToPicture(hdib)
End If
```

DIB_FromPicture

```
HANDLE DIB_FromPicture(Picture pic)
```

Create a DIB (Device-Independent Bitmap) from a Picture. **The Picture itself is unchanged.** This only works if the Picture contains a bitmap – not an icon or metafile.

Drawing Text into DIBs

DIB_DrawText

```
void DIB_DrawText(HANDLE hdib, const char *pzText, int x, int y,
int w, int h)
```

Draw text into the specified DIB image, inside the specified rectangle. *All the coordinates are in pixels.* The y-coordinate is measured down from the top, so (0,0) is the upper-left corner of the image. A width or height of -1 means 'as much as needed'. When you first use this function, we recommend using w = h = –1.

The text color, typeface, character height, rotation angle and format can be set with the functions below. The default text settings are:

Normal (plain) style
flush left, push to top
character height: 14 pixels
rotation angle: 0 degrees (horizontal)
color: black
font: Arial

Example:

```
DIB_SetTextColor(255, 0, 0);
DIB_SetTextFormat(EZT_TEXT_RIGHT+EZT_TEXT_BOTTOM);
DIB_DrawText(hdib, "Captured 2003.07.08 11:52", 0, 0,-1,-1);
```

This will draw the annotation in the bottom-right corner of the image in bright red.

Note: Anisotropic images

When drawing text into an image that has different DPI in X and Y, such as a digital fax file, the text height is reinterpreted as a physical height relative to the higher DPI value. In other words, if you set text height=25 and draw text into an image with 200 DPI horizontal and 96 DPI vertical, the text height is interpreted as 25/200 = 0.125 inches, and text is drawn correctly proportioned to be 0.125 in. high no matter how it is oriented. In this example, horizontal text would be drawn 25 pixels high and would print out 0.125 inches high. Text drawn sideways (angles of 90 or 270) will be drawn in the image 0.125 * 96 = 12 pixels high, which causes it to print and display at 12/96 = 0.125 inches high, the same as horizontal text.

DIB_SetTextHeight

```
void DIB_SetTextHeight(int nH)
```

Set the character height in pixels (image rows) for subsequent calls to DIB_DrawText. If you need to set the text height in physical units (inches) convert as follows:

nH = round_to_nearest_integer(HeightInInches * DIB_Yresolution(hdib))

DIB_SetTextColor

```
void DIB_SetTextColor(int R, int G, int B)
```

Set the text color for subsequent calls to DIB_DrawText.

DIB_SetTextAngle

```
void DIB_SetTextAngle(int nDegrees)
```

Set the text orientation for subsequent calls to DIB_DrawText, in degrees of rotation clockwise from horizontal. Only multiples of 90 degrees are supported. Negative values (representing counter-clockwise rotation) are accepted.

DIB_SetTextFace

```
void DIB_SetTextFace(const char *pzFace)
```

Set the text typeface (text font) for subsequent calls to DIB_DrawText. Default is "Arial". The fonts that are absolutely universal on Windows (including 95 and NT):

Arial
Courier New
Lucida Console
MS Sans Serif
Times New Roman
Symbol: ΑΒΧΔΕΦΓ
WingDings: ♋♌♍♎♏♐♑

A machine that has Internet Explorer installed will have additional fonts:

Arial Black
Comic Sans MS
Georgia
Impact
Trebuchet MS
Verdana

DIB_SetTextFormat

```
void DIB_SetTextFormat(int nFlags)
```

Sets the alignment and formatting of text for for subsequent calls to DIB_DrawText. These format attributes can be added or OR'ed together:

SetTextFormat Flags

Named Constant	Value	Meaning / Effect
EZT_TEXT_NORMAL	0x0000	'plain' text style, like this
EZT_TEXT_BOLD	0x0001	**bold**
EZT_TEXT_ITALIC	0x0002	*italic*
EZT_TEXT_UNDERLINE	0x0004	underlined
EZT_TEXT_STRIKEOUT	0x0008	~~strikeout~~
EZT_TEXT_BOTTOM	0x0100	Align bottom of text to bottom of rectangle
EZT_TEXT_VCENTER	0x0200	Center text vertically within rectangle
EZT_TEXT_TOP	0x0000	Align top of text to top of rectangle
EZT_TEXT_LEFT	0x0000	Align text to the left side
EZT_TEXT_CENTER	0x1000	Center text between left and right sides
EZT_TEXT_RIGHT	0x2000	Align text to the right side
EZT_TEXT_JUSTIFY	0x0800	Stretch text out to left *and* right sides
EZT_TEXT_WRAP	0x4000	Break lines that are too wide to fit

DIB Transformations & Drawing

DIB_DrawLine

```
void DIB_DrawLine(HDIB hdib,          // DIB to draw into
                  int x1, int y1,     // starting point
                  int x2, int y2,     // ending point
                  int R, int G, int B)  // color
```

Draw a straight line in an image (DIB) from the starting point to the ending point, using the nearest available match to the specified color. R, G, B are interpreted as 8-bit red/green/blue values 0..255. The coordinates are in pixels, with (0,0) being the upper-left corner of the image. If hdib is a B&W, grayscale, or palette image, the mathematically nearest representable color to (R,G,B) is used.

DIB_Fill

```
void DIB_Fill(HANDLE hdib, int R, int G, int B)
```

Fill all the pixels of the DIB with the specified color. R, G, B are interpreted as 8-bit red/green/blue values 0..255. If hdib is a B&W, grayscale, or palette image, the mathematically nearest representable color to (R,G,B) is used.

DIB_Negate

```
void DIB_Negate(HANDLE hdib)
```

Negates all the pixels in the DIB. Note that the color table if any is left untouched – this call will not have the desired effect on an indexed-color image.

DIB_AdjustBC

```
void DIB_AdjustBC(HDIB hdib, int nB, int nC)
```

Adjust the brightness and/or contrast of the image.
nB and nC are -1000 to 1000, with a value of 0 meaning 'no change'.
Positive nB pushes all pixels toward white, negative toward black.
Positive nC pushes all pixels away from mid-value, toward black and white.
Negative nC pushes all pixels toward the mid-value.
Works on grayscale, RGB, CMY(K) images - no effect on B&W and palette.

DIB_FlipVertical

```
void DIB_FlipVertical(HANDLE hdib)
```

Flips the image in the y-direction – turns it upside down.

DIB_FlipHorizontal

```
void DIB_FlipHorizontal(HANDLE hdib)
```

Flips the image in the x-direction – mirror-images it.

DIB_Rotate180

```
void DIB_Rotate180(HANDLE hdib)
```

Turns the image 180°.

DIB_Rotate90

```
HANDLE DIB_Rotate90(HANDLE hOld, int nSteps)
```

Return a *copy* of hOld rotated clockwise nSteps * 90°. If nSteps is 0, the result is a copy of hOld. Negative values of nSteps rotate counterclockwise. Note that *hOld is not destroyed* so you need to DIB_Free it if you are done with it.

DIB Scaling, Resampling & Format Conversion

DIB_ScaledCopy

```
HANDLE DIB_ScaledCopy(HANDLE hOld, int w, int h)
```

Create and return a **copy** of hOld scaled (resampled) to have width w and height h. This only works on 24-bit color and 8-bit grayscale images, other input will cause the function to fail and return NULL. Don't forget to DIB_Free the old DIB when you are done with it. This operation is sometimes called *resampling* because you have the same image, but with a different number of *samples* of its color values.

DIB_Resample

```
HDIB DIB_Resample(HDIB hOld, double xdpi, double ydpi)
```

Return a new image that is a copy of the old image, but resampled to the specified resolution. *Resampling* is the technical term for recomputing the pixels of an image, when you want to change the number of pixels in the image but not the physical size (like 8.5" x 11").

If you resample from 300DPI to 100DPI, you will have 1/3 as many rows, 1/3 as many columns, 1/9 as many pixels - but the pixels will be marked in the image as being 3 times as 'wide' and 'tall' - so the physical size of the image stays the same. This is the same as DIB_ScaledCopy (above), just looked at in a different way.

DIB_Resample will fail if the input image has either resolution <= 0, or if xdpi or ydpi is <= 0. It can also fail from insufficient memory.

Remember to DIB_Free the old DIB when you are done with it.

DIB_Thumbnail

```
HANDLE DIB_Thumbnail(HANDLE hdib, int MaxWidth, int MaxHeight)
```

Return an image (Dib) containing a copy of hdib, scaled so that its width is no more than MaxWidth, and height is no more than MaxHeight. Can accept *any* image produced by EZTwain. B&W images are converted to grayscale thumbnails. Remember to DIB_Free the original image and the thumbnail, when you are done using them.

DIB_SimpleThreshold

```
HDIB DIB_SimpleThreshold(HDIB hdib, int nT)
```

Returns a B&W copy of the given image using the given threshold. The gray value (0..255) of each input pixel is calculated: If the value < nT that pixel is black (0) in the returned image, if the value >= nT the pixel becomes white in the returned image. Remember to DIB_Free each image when you are done using it.

DIB_SmartThreshold

```
HDIB DIB_SmartThreshold(HDIB hdib)
```

Returns a B&W copy of the given image, using a 'smart' thresholding algorithm. This function examines the entire image and chooses a threshold value that is 'optimal' in some sense, for text and line-art. Remember to DIB_Free each image when you are done using it.

DIB_ConvertToPixelType

```
HANDLE DIB_ConvertToPixelType(HANDLE hdib, int nPT)
```

Takes a DIB handle and a pixel-type code (see Pixel Types, p 4) and returns a copy of the input DIB, converted into the specified pixel type. The input DIB is not affected, and must be freed with DIB_Free when no longer needed.

This function can be used to expand B&W images into grayscale, to convert between RGB and CMY or CMYK formats, to convert color scans to grayscale, grayscale to B&W, and so forth.

When converting to B&W, the image is thresholded using a 'smart threshold' - see DIB_SmartThreshold above. When converting a color image to TWPT_PALETTE, an optimized color table is computed, and the image is rendered into that set of colors with a 'random dither' technically known as *error diffusion.*

DIB_ConvertToFormat

```
HANDLE DIB_ConvertToFormat(HDIB hOld, int nPT, int nBPP)
```

Create and return a new DIB containing the hOld image converted to the specified pixel type and bits per pixel. Similar DIB_ConvertToPixelType but allows for non-standard depth in the output, such as 4-bit/pixel grayscale, or 16-bit/sample RGB. Unsupported and impossible combinations cause a NULL return.

DIB_ScaleToGray

```
HANDLE DIB_ScaleToGray(HDIB hdibOld, int r)
```

Create and return a new grayscale DIB by averaging each r x r pixel square of hdibOld to create each output pixel. The output image has 1/r times the width and height, and resolution, of the input image. Works well to convert B&W images to lower-resolution grayscale images, which are not as 'crisp' but are smoother, look better when scaled, and can be JPEG-compressed.

DIB Block Copy and Masking

DIB_RegionCopy

```
HANDLE DIB_RegionCopy(HANDLE hOld, int x, int y, int w, int h,
int fill)
```

Create and return a new DIB that is a copy of a rectangular region of hOld. The copied region is w pixels wide, h pixels high, starting at (x, y) in the hOld image, where (0,0) is the upper-left corner of hOld, visually. Pixels that don't fit into the new DIB are discarded. If the new DIB is taller or wider than the old, the new pixels on the right and bottom are filled with bytes = *fill*. Common values for fill are:

-1 (or 255 or 0xFF)	fills with 1's producing white
0	which produces black fill.

This function is useful for *cropping* or *extending* an image.

DIB_Blt

```
void DIB_Blt(
      HANDLE hdibDst,        // DIB destination
      int dx, int dy,        // destination (x,y)
      HANDLE hdibSrc,        // DIB source
      int sx, int sy,        // where to start in source
      int w, int h,          // width and height
      unsigned uRop)         // operation to apply
```

Copy pixels from hdibSrc into hdibDst, starting at (dx,dy) in the destination, and (sx,sy) in the source, and transferring a rectangular region w columns by h rows. Any pixels that fall outside the actual bounds of the source and destination DIBs are ignored. Put another way, the coordinates and sizes are clipped to boundaries of the actual DIBs. The operations available are:

EZT_ROP_COPY	0
EZT_ROP_OR	1
EZT_ROP_AND	2
EZT_ROP_XOR	3

DIB_BltMask

```
void DIB_BltMask(
      HANDLE hdibDst,        // DIB destination
      int dx, int dy,        // destination (x,y)
      HANDLE hdibSrc,        // DIB source
      int sx, int sy,        // where to start in source
      int w, int h,          // width and height
      unsigned uRop,         // operation to apply (see note below)
      HANDLE hdibMask) // the 'mask'
```

Like DIB_Blt, but hdibMask contains an 8-bit mask. hdibMask must be the same size as hdibSrc, and must be 8-bits per pixel. For each pixel of the affected region, if D is the destination pixel, S is the source, and M is the mask:

D = (M / 255) * S + (1 – M / 255) * D

So a mask value of 255 (usually white) causes the source pixel to replace the old destination value, a mask value of 0 leaves the destination value unchanged, and in between the source and destination are blended according to the mask value.

Note: Currently only EZT_ROP_COPY is supported for the uRop parameter.

DIB_PaintMask

```
void DIB_PaintMask(
      HANDLE hdibDst,        // DIB destination
      int dx, int dy,        // starting destination (x,y)
      int R, int G, int B,   // color to paint with
      int sx, int sy,        // starting mask (x,y)
      int w, int h,          // width and height
      unsigned uRop,         // operation to apply (see note below)
      HANDLE hdibMask) // the 'mask'
```

Like DIB_BltMask, but applies a solid color. hdibMask must be 8-bits deep. For each pixel of the affected region, if D is the destination pixel value, P is the paint color, and M is the mask:

D = (M / 255) * P + (1 – M / 255) * D

A destination pixel D is only affected if it exists in hdibDst, falls within the rectangle specified by (dx, dy, w, h), and if the corresponding mask pixel exists in hdibMask. A width value (w) of –1 is interpreted as 'as wide as possible', similarly for height (h). If you paint into a grayscale DIB, the paint color is converted to a gray value if necessary. If you paint into a B&W DIB, the paint color is converted to gray, then to whichever is closest: black, or white.

Note: Currently only EZT_ROP_COPY is supported for the uRop parameter.

Working with a DIB through a DC

DIB_OpenInDC

```
int DIB_OpenInDC(HANDLE hdib, HDC hdc)
```

Create a temporary copy of the DIB and select it into the specified DC (Device Context). Allows use of any GDI function to draw through the DC into the image. The drawing actually takes place in a temporary bitmap called a *DIBSection*, the result are only copied back into your hdib when you call DIB_CloseInDC – see below.

Only one DIB can be open this way at a time: You cannot nest DIB_OpenInDC. A second call to DIB_OpenInDC without an intervening DIB_CloseInDC will display an error message and return –4.

Return values:

0	Success
-1	Could not lock the hdib – probably not a valid handle
-2	CreateDIBSection failed: hdc is invalid, hdib is not a DIB handle, or insufficient memory.
-3	Unable to select the DIBSection into the hdc (Unknown cause)
-4	Nested call – two calls to OpenInDC without a call to CloseInDC.

DIB_CloseInDC

```
void DIB_CloseInDC(HANDLE hdib, HDC hdc)
```

Call this function exactly once for each call to DIB_OpenInDC. It has no effect at any other time. Copies the image from the DC back into the DIB and detaches it from the DC. Example:

```
// This draws a disc of reversed color in the upper-left corner of the
// image (hdib), using the GDI 'Ellipse' function in exclusive-OR mode.
HDC hdc = CreateCompatibleDC(NULL);
if (hdc) {
   if (0==DIB_OpenInDC(hdib, hdc)) {
      SetROP2(hdc, R2_XORPEN);          // set 'exclusive-or' mode
      Ellipse(hdc, 4, 4, 132, 132);     // Draw a filled circle
      DIB_CloseInDC(hdib, hdc);
   }
   DeleteDC(hdc);
}
```

DIBs: Automatic Image Improvement

DIB_AutoCrop

```
HANDLE DIB_AutoCrop(HDIB hOld, int nOptions)
```

Return a copy of the image in hOld, with the surrounding border of uniform color (if there is one) cropped off. Of course this will normally change the dimensions of the image - the pixel type and depth are not changed.
After this call, remember to DIB_Free(hOld) if you don't need it.
nOptions is currently unused and must be 0 (zero).

DIB_GetCropRect

```
BOOL DIB_GetCropRect(HDIB hdib, int nOptions,
                     int *x, int *y, int *w, int *h)
```

Returns a suggested crop rectangle to remove blank or unused border from the image. The returned rectangle is defined by an upper-left point and a width and height, in pixels. (Precisely the arguments needed by DIB_RegionCopy.) As usual, y and h are measured *down* from the top of the image.

nOptions is currently unused and must be 0.

DIB_AutoCrop uses this function to decide what to crop.

A return of FALSE means no crop rectangle was found - generally this means that the image has content that extends to the edges, or has no definite borders of dark color. For convenience, when this function returns FALSE it sets x, y, w and h to specify the entire image.

DIB_AutoDeskew

```
HANDLE DIB_AutoDeskew(HANDLE hOld, int nOptions)
```

Returns a copy of the image in hOld, possibly 'deskewed'.
If it can be determined that the input image is consistently skewed (rotated by a small angle) then the returned image is rotated to eliminate that skew.
After this call, remember to DIB_Free(hOld) if you don't need it.
The depth and pixel type of the image are not changed.
The dimensions of the returned image may be slightly changed.
nOptions is currently unused and must be 0 (zero).

DIB_DeskewAngle

```
double DIB_DeskewAngle(HANDLE hdib)
```

Compute and return the small clockwise rotation that would best deskew (vertically align) the given image. The returned angle is in radians, which may be negative or positive. Only rotations in the range ±4° are considered. A value < -9.0 means that

an optimal rotation could not be determined. A deskew angle in the range ±0.001 can probably be ignored - the image is already nearly perfectly upright.

DIB_AutoContrast

```
int DIB_AutoContrast(HANDLE hdib)
```

Automatically adjust the brightness and contrast of an image to 'improve' it. For bimodal images, where the original material appears to be 'perceptually black and white', this function will adjust the brightness and contrast to make the dominant light color into white, and the primary dark tone into black. For other images, this function evaluates whether the images is using the available tonal range, and if not attempts to adjust brightness and contrast to expand the image's tonal range. DIB_AutoContrast has no effect on B&W images.

DIB_MedianFilter

```
void DIB_MedianFilter(HDIB hdib, int W, int H, int nStyle)
```

Apply a median filter to an image using an W x H neighborhood.
The parameter nStyle is currently ignored, but should be 0 for future compatibility.

The median filter is effective at removing speckle noise from color and grayscale images, because it smooths out pixels that differ radically in value from their neighbors.

DIBs: Image Analysis

DIB_IsBlank

```
BOOL DIB_IsBlank(HDIB hdib, double dDarkness)
```

Return TRUE(1) if the DIB has less than dDarkness fraction of 'dark' pixels, FALSE(0) otherwise. A typical value of dDarkness would be 0.02 which means 2% dark pixels. A page with less than 2% dark pixels is probably blank.
See How To: Skip Blank Pages, p 20.

DIB_Darkness

```
double DIB_Darkness(HDIB hdibFull)
```

Returns the fraction of an image that consists of 'dark' pixels i.e. pixels that would be black if the image was converted to B&W using a smart thresholding. (See DIB_SmartThreshold p. 57 for more details.) A return of 0.0 means none, 1.0 means all. A typical office document is 0.02 (2%) to 0.32 (32%) dark pixels. This function is used by DIB_IsBlank to decide if an image is blank.

DIB_GetHistogram

```
void DIB_GetHistogram(HANDLE hdib, int nComp, int nHisto[256])
```

This function computes a histogram of the given DIB. The third parameter must be an array of 256 integers (32-bit or 64-bit depending on whether you are using the x86 or x64 version of EZTwain) – it need not be initialized, it is output-only. When DIB_GetHistogram returns, each entry nHisto[v] contains the number of pixels in hdib that have a value of v in the specified component. This function works on B&W, grayscale, RGB, and Palette images.

Component/Channel Codes

Symbol	C	Description
COMPONENT_GRAY	0	Grayscale equivalent or lightness
COMPONENT_RED	1	Red component / red channel
COMPONENT_GREEN	2	Green component
COMPONENT_BLUE	3	Blue component
COMPONENT_SAT	4	Saturation (as in HSB color model)
COMPONENT_HUE	5	Hue (as in HSB/HSV/HSL color models)
COMPONENT_LUMINANCE	0	synonym for COMPONENT_GRAY

DIB_Avg/DIB_AvgRegion/DIB_AvgRow/DIB_AvgColumn

```
double DIB_Avg(HDIB hdib, int nComp)
double DIB_AvgRegion(HDIB hdib,int nComp,int x,y,w,h)
double DIB_AvgRow(HDIB hdib, int nComp, int y)
double DIB_AvgColumn(HDIB hdib, int nComp, int x)
```

Average the values of pixels in an image, region, row or column, and return the resulting value between 0 and 255.

Note that row 0 (y=0) is the visually top-most row of an image.
Averages either intensity (brightness) or individual color channels.
See component codes above, for DIB_GetHistogram.
All image formats are normalized so that white = 255.0 and black = 0, even for 1-bit B&W or 16-bit grayscale or color images.

DIB_ComponentCopy

```
HDIB DIB_ComponentCopy(HDIB hdib, int nComponent)
```

Extract and return an image containing one component (channel) of the input image.
See *component codes* under DIB_Histogram.
The returned image is an 8-bit grayscale image containing the specified channel of the input image, with the same width, height, and DPI as the input image.

Caution: *Future versions may return a 16-bit deep image if given a 16 bit/channel input image.*

DIBs: Miscellaneous

DIB_SetColorCount

```
void DIB_SetColorCount(HANDLE hdib, int n)
```

DIB_SwapRedBlue

```
void DIB_SwapRedBlue(HANDLE hdib)
```

For 24-bit DIB only, exchange R and B components of each pixel.

DIB_CreatePalette

```
HPALETTE DIB_CreatePalette (HANDLE hdib)
```

Create and return a logical palette to be used for drawing the DIB.
For 1, 4, and 8-bit DIBs the palette contains the DIB color table.
For 24-bit DIBs, a default halftone palette is returned.

DIB_Lock

```
BITMAPINFOHEADER* DIB_Lock(HANDLE hdib)
```

Lock the given DIB handle and return a pointer to the header structure. Technically, increments the lock count of hdib and returns its address.

DIB_Unlock

```
void DIB_Unlock(HANDLE hdib)
```

Unlock the given DIB handle (decrement the lock count.)
When the number of Unlocks = the number of Locks, any pointers into the DIB should be presumed invalid.

DIBs: Clipboard Functions

DIB_PutOnClipboard

```
int DIB_PutOnClipboard(HANDLE hdib)
```

Place the DIB on the clipboard (format CF_DIB.)
Important: After this call, the clipboard owns the DIB and *you do not* - you should set your copy of the DIB handle to NULL (0) and attempt no further operations on that DIB. Treat this call just as you would a call to **DIB_Free**.
Returns 1 = success, 0 = failure.

DIB_CanGetFromClipboard

```
BOOL DIB_CanGetFromClipboard(void)
```

Return 1 if there is something on the clipboard that can be delivered as a DIB (by DIB_GetFromClipboard below.) Returns 0 if not. If you are implementing 'paste', call this function to enable and disable the Paste command.

DIB_GetFromClipboard
DIB_FromClipboard

```
HANDLE DIB_GetFromClipboard(void)
HANDLE DIB_FromClipboard(void)
```

Create and return a DIB with the contents of the clipboard.
Used to implement a Paste function for images. Returns NULL in case of error, or if no image on clipboard. See DIB_CanGetFromClipboard above.

Functions – Printing

Configuration

DIB_SpecifyPrinter

```
int DIB_SpecifyPrinter(string pzPrinterName)
```

Specify the printer to be used when printing to the 'default printer', as in DIB_PrintNoPrompt below. In other words, this overrides the user's default printer choice.
Calling this function with NULL or an empty string tells EZTwain to return to using the system default printer as the default printer.

DIB_EnumeratePrinters

```
int DIB_EnumeratePrinters()
```

Enumerate the available printers and return the count of printers found. A return value < 0 indicates some serious internal error. After this call, use DIB_PrinterName or DIB_GetPrinterName (below) to get the names of the available printers. This call can take several seconds - possibly more on some versions of Windows, depending on whether there are remote printers in the list.

DIB_PrinterName / DIB_GetPrinterName

```
char* DIB_PrinterName(int i)
int DIB_GetPrinterName(int i, LPSTR PrinterName)
```

Get the name of the *ith* available printer. DIB_PrinterName returns the name as a string. DIB_GetPrinterName copies it into the specified character array (string buffer). In most languages, you will need to allocate and/or initialize the string variable to be 256 characters.

DIB_SetPrintToFit / DIB_GetPrintToFit

```
void DIB_SetPrintToFit(int nYes)
int DIB_GetPrintToFit()
```

Get or set the print-to-fit flag.
When the print-to-fit flag is non-zero, EZTwain reduces the size of printed images to fit within the printer page. This only affects images that are too large to fit on the page. By default, this flag is FALSE (0)

Single-Page Printing

DIB_Print

```
int DIB_Print(HANDLE hdib, string pzJobname)
```

Display the standard Print dialog, and if OK'd by user, print the DIB on the user-selected printer. By default, prints the DIB at 'physical size' - the DIB resolution values are used to convert the width and height from pixels to physical units (e.g. inches.) If the DIB has resolution values of 0, 72 DPI is assumed. However, if the print-to-fit flag is set (see DIB_SetPrintToFit above) any image too large to print on the printer page is scaled smaller until it fits. The image is always printed centered on the page.

The 2nd parameter is a string that appears in the print queue; If it is NULL or the empty string, the application title is used (See TWAIN_SetAppTitle.)

DIB_PrintNoPrompt

```
int DIB_PrintNoPrompt(HANDLE hdib, string pzJobname)
```

Like DIB_Print, but does not prompt the user. The image is printed on the default printer with default print settings.

Multipage Printing from a File

DIB_PrintFile (alias TWAIN_PrintFile)

```
int DIB_PrintFile(string file, string jobname, BOOL bNoPrompt)
int TWAIN_PrintFile(string file, string jobname, BOOL bNoPrompt)
```

Print the specified file with the specified job name.
If the filename is null or empty, the user is prompted to select a file.
If the jobname is null or empty, the actual filename is used as the jobname.
If bNoPrompt is non-zero (True) the job is sent to the default printer.
If bNoPrompt is zero (False) the user is prompted with the standard Print dialog.
Return values:

0	success
-1	user cancelled Open File dialog
-2	could not open/access printer
-3	error reading from file
-4	printing output error
-10	user cancelled Print dialog

Multipage Printing – DIBs

If you don't have your images in a file, you can print multipage documents from memory using these functions. DIB_PrintArray prints an array of images as a single print job. Or you can compose a print job yourself: Call DIB_PrintJobBegin to start the job, call DIB_PrintPage with each page image and call DIB_PrintJobEnd when done. Try to always call DIB_PrintJobEnd, even in event of an error: Otherwise various things are left in an undesirable state.

DIB_PrintArray

```
int DIB_PrintArray(HDIB hdibs[], int nCount,
                   string Jobname, BOOL bNoPrompt)
```

Prints the first nCount images in the hdibs array, under the given print-job name. If the job-name parameter is NULL or the empty string, the application title is used. If bNoPrompt is TRUE(non-zero), the print job is sent directly to the default printer. If bNoPrompt is FALSE(0), the user is prompted with the standard print dialog. Return value is same as DIB_Print (above).

DIB_PrintJobBegin

```
int DIB_PrintJobBegin(string pzJobname, BOOL bUseDefaultPrinter)
```

Begins a multipage print job. Jobname is the name of the print job: This appears in the print queue , and in some environments it is printed on a job-separator page ahead of the job. If Jobname is null or empty, the application title is used. (See TWAIN_SetAppTitle)

If bUseDefaultPrinter is non-zero (true) the default printer is used, otherwise the user is prompted to select the printer. If a print job is open, DIB_PrintJobEnd() is called to close it. Return values:

0	success
-2	could not open/access printer
-4	printing output error
-10	user cancelled Print dialog

DIB_PrintPage

```
int DIB_PrintPage(HDIB hdib)
```

Print a page as part of the current job. See DIB_Print for more details.
Return values:

0	success
-3	the DIB is null or invalid
-4	printing output error
-5	no print job is open

DIB_PrintJobEnd

```
int DIB_PrintJobEnd(void)
```

End the current print job and release it for printing.
(Some environments will start printing as soon as the first page is available.)

Return values:

0	success
-4	printing output error
-5	no print job is open

Functions – Barcode Recognition

Introduction

EZTwain barcode recognition is based on a multi-engine architecture with a specific set of supported engines in each release. You can enumerate the supported or available engines, select an engine, and use that engine to search scanned or loaded images for barcodes.

Some barcode jargon you may encounter:

A *symbology* is a style of barcode, such as *Code 128*, defined by a set of rules for bar widths, heights, clearance, character encoding, and error detection/correction.

A *symbol* is what the rest of the world calls 'a barcode'. This can be confusing – in barcode talk, a symbol is an entire barcode. Also called a *patch*, although that tends to be used more for 2D barcodes.

The general paradigm for analyzing barcodes in a scanned or loaded image is this:

1. To enumerate the defined engines, call BARCODE_EngineName(i) for i = 1, 2, ... until it returns the empty string.
2. Select the desired engine using BARCODE_SelectEngine. Note also BARCODE_IsEngineAvailable.
3. Select the possible orientations for that should be searched for barcodes, using BARCODE_SetDirectionFlags.
4. You can use BARCODE_ReadableCodes to determine which barcode types are recognizable by the selected engine.
5. If you know the approximate location of your barcodes, you can improve speed and avoid 'false positives' (finding barcodes you don't want to find) by setting a recognition zone with BARCODE_SetZone.
6. Call BARCODE_Recognize, passing it the handle of the image to search, the maximum number of barcode patches to find, and a mask of the barcode types (symbologies) to look for. If this function finds any barcodes, it returns a positive integer = the count of symbols (barcodes) found.
7. If *n* barcodes were found, use BARCODE_Text, BARCODE_Type, BARCODE_GetRect, and BARCODE_GetText or BARCODE_Text to obtain details about each barcode, passing in an index from 0 to *n*-1.

If you would like to learn more about the theory and practice of barcoding, we recommend *The Bar Code Book* by Roger C. Palmer (available through Amazon.com). There are also some helpful links on our website, at
http://www.eztwain.com/barcode.htm

Supported Barcode Engines

As of version 3.48, EZTwain Pro supports five barcode engines. Contact Technical Support if you are interested in using another barcode engine.

1. EZTwain Native Barcode Engine

A very limited built-in barcode engine which recognizes only horizontal and vertical 3-of-9 symbols. This engine is selected by default.

For more advanced barcode recognition, EZTwain can detect and use the following *third party* barcode engines, which must be licensed from their respective vendors.

2. LEADTOOLS Linear 1D Symbols Engine

http://www.leadtools.com/SDK/Document/Document-Addon-Barcodelinear1D.htm

The LeadTools engine seems to be the fastest and one of the most accurate, handles slanted (skewed) barcodes well, and is relatively expensive. You must license the LeadTools Raster Imaging SDK as well as the Linear 1D Barcode SDK, plus purchase a run-time license for each machine that will use barcode recognition. Check the website above for the latest information.

The following files are required when using LeadTools 15 barcode recognition:

```
Ltkrn15u.dll          Ltbar15u.dll          Ltdis15u.dll
Ltimgcor15u.dll       Ltimgutl15u.dll       Ltbar415u.dll
```

For LeadTools 16, looks like you need (at least)

```
Ltkrnu.dll            Ltbaru.dll            Ltdisu.dll
Ltimgcoru.dll         Ltimgutlu.dll         Ltbar4u.dll
```

Note: All Unicode versions of LEADTOOLS require Microsoft Unicode Layer for Windows (UNICOWS.DLL) in order to function on Windows 95/98/Me. You can obtain MSUL from: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/anch_mslu.asp.

Note: LEADTOOLS Requires the following Microsoft C/C++ Runtime files to be distributed in the application's PATH:

Win32 Platforms:	**x64 Platforms:**
msvcr80.dll msvcp80.dll Microsoft.VC80.CRT.manifest mfc80u.dll Microsoft.VC80.MFC.manifest MFC80ENU.dll Microsoft.VC80.MFCLOC.manifest	msvcr80.dll msvcp80.dll Microsoft.VC80.CRT.manifest mfc80u.dll Microsoft.VC80.MFC.manifest MFC80ENU.dll Microsoft.VC80.MFCLOC.manifest

NOTE: that the filenames are the same for Win32 and x64. However, the actual

binares are different. For more information about distributing the Microsoft C/C++ runtime files, refer to:
http://msdn2.microsoft.com/en-us/library/ms235291.aspx

The Microsoft C Runtime business is quite a mess, see for example:

http://www.codeproject.com/cpp/vcredists_x86.asp

3. Black Ice 1D Barcode Engine

See http://www.blackice.com/barcodeLinear1D.htm
The Black Ice engine can be licensed with a one-time license fee, and seems fast when dealing with horizontal and vertical symbols (less than 6° of skew). However we experienced much slower scanning when searching for diagonal barcodes. Some customers have reported that they found the Black Ice engine a bit more accurate than the LeadTools and Axtel engines.

EZTwain looks for and loads "BiBrw1D.dll" - check the Black Ice SDK for information about other DLLs required by BiBrw1D.dll.

Note: When setting up the Black Ice barcode engine, don't forget to copy the file "impBarcode.adl".

4. Axtel AX-4 Linear Barcode Engine

This barcode engine is no longer being licensed by Axtel. However, if you have a copy of AXBAR32.DLL, EZTwain Pro can use it. Select EZBAR_ENGINE_AXTEL.

5. Inspirant "INBarcodeOCR" Linear Barcode Engine

Available from Inspirant (www.inspirant.de) this engine is contained in a single file, INBarcodeOCR.DLL. It supports EAN13, EAN8, UPCA, Code39, Code128, Interleaved 2-of-5, and UCCEAN128.

License it from Inspirant, place the DLL in the same folder as your copy of Eztwain4.dll, or in System32/SysWOW64, and select it using:

```
BARCODE_SelectEngine(EZBAR_ENGINE_INBARCODE)
```

BARCODE_IsAvailable

```
BOOL BARCODE_IsAvailable(void)
```

TRUE(1) if some barcode recognition is available (the necessary components and DLLs are present and loadable). Returns FALSE(0) otherwise. For any barcode services to be available, the EZT4Symbol.dll must be present and loadable. Some developers do not include this DLL in their configuration, and in this case BARCODE_IsAvailable will return FALSE (0).

The barcode manager and built-in engine are implemented in the helper library EZT4Symbol.dll. EZT4Symbol.dll must be installed next to Eztwain4.dll – i.e. In the same folder. Additional barcode engine DLLs, if any, must be placed where LoadLibrary can find them: Alongside the Eztwain4.dll, in the System32/SysWOW64 folder, or somewhere on the PATH.

BARCODE_IsEngineAvailable
BARCODE_SelectEngine
BARCODE_SelectedEngine
BARCODE_EngineName

```
BOOL BARCODE_IsEngineAvailable(int nEngine)
BOOL BARCODE_SelectEngine(int nEngine)
int BARCODE_SelectedEngine(void)
string BARCODE_EngineName(int nEngine)
```

These four functions allow you to detect which engines are available, to select the engine to use for recognition, and to get human-readable engine names.

Barcode Engine Codes

Symbol	Code	Description
EZBAR_ENGINE_NONE	0	'null' barcode engine – no recognition.
EZBAR_ENGINE_NATIVE	1	built-in Code 3-of-9 engine
EZBAR_ENGINE_AXTEL	2	Axtel AX-4 engine
EZBAR_ENGINE_LEADTOOLS	3	LEADTOOLS engine (ltbar15u.dll)
EZBAR_ENGINE_BLACKICE	4	Black Ice engine (BiBrw1D.dll)
EZBAR_ENGINE_LEADTOOLS16	5	LEADTOOLS V16 barcode engine
EZBAR_ENGINE_INBARCODE	6	Inspirant INBarcodeOCR

Note 1: 'engine 0' is the null engine – it does nothing and recognizes no barcode types.

Note 2: In EZTwain 3, EZBAR_ENGINE_NATIVE was EZBAR_ENGINE_DOSADI.

BARCODE_ReadableCodes

```
int BARCODE_ReadableCodes(void)
```

Returns a mask of the barcode types (symbologies) recognized by the currently selected barcode engine.

Barcode Types (Symbologies)

Barcode Types (Symbologies)	Value (hex)	Value (decimal)
EZBAR_EAN_13	0x00000001L	1
EZBAR_EAN_8	0x00000002L	2
EZBAR_UPCA	0x00000004L	4
EZBAR_UPCE	0x00000008L	8
EZBAR_CODE_39	0x00000010L	16
EZBAR_CODE_128	0x00000040L	64
EZBAR_CODE_I25	0x00000080L	128
EZBAR_CODA_BAR	0x00000100L	256
EZBAR_UCCEAN_128	0x00000200L	512
EZBAR_CODE_93	0x00000400L	1024
EZBAR_ANY	0xFFFFFFFFL	-1

BARCODE_TypeName

```
String BARCODE_TypeName(int nType)
```

Returns a human-readable name for the specified barcode type/symbology.

BARCODE_SetDirectionFlags
BARCODE_GetDirectionFlags
BARCODE_AvailableDirectionFlags

```
BOOL BARCODE_SetDirectionFlags(int nDirFlags)
int BARCODE_GetDirectionFlags(void)
int BARCODE_AvailableDirectionFlags(void)
```

Define the directions the engine will scan for barcodes. The default is *left-to-right*. Scanning for barcodes in multiple directions will slow the recognition process. BARCODE_SetDirectionFlags will return TRUE if completely successful, FALSE if any requested direction is invalid or not supported. As a special case, setting the direction flags to -1 is interpreted as "select all supported directions."

The native barcode engine does not support recognition of diagonal (highly skewed) symbols.

Barcode Direction Flags

Barcode Direction Flags	Value (can be OR'd together)
EZBAR_LEFT_TO_RIGHT	1
EZBAR_RIGHT_TO_LEFT	2
EZBAR_TOP_TO_BOTTOM	4
EZBAR_BOTTOM_TO_TOP	8
EZBAR_DIAGONAL	16
Common combinations	
EZBAR_HORIZONTAL	3
EZBAR_VERTICAL	12

BARCODE_SetZone
BARCODE_NoZone

```
void BARCODE_SetZone(int x, int y, int w, int h)
void BARCODE_NoZone()
```

BARCODE_SetZone restricts subsequent barcode recognition to a rectangular zone in each image. The rectangle is defined by x,y,w,h: x = pixels from left edge, y = pixels from top edge, w = width in pixels, h = height in pixels.

BARCODE_NoZone restores the default condition, in which barcode recognition is performed throughout each image.

BARCODE_Recognize

```
int BARCODE_Recognize(HDIB hdib, int nMaxCount, int nType)
```

Find and recognize barcodes in the given image.
Don't look for more than nMaxCount barcodes (-1 means 'any number').
Expect barcodes of the specified type (-1 means 'any supported type')
You can add or 'or' together barcode types, to tell the recognizer to look for more than one symbology. Return values:

>0	n barcodes found
0	no barcodes found
-1	barcode services not available.
-3	invalid or null image

Not surprisingly, recognition slows down as you allow more barcodes to be found, and as you allow more symbologies to be recognized.

BARCODE_Type

```
int BARCODE_Type(int n)
```

Return the type of the nth barcode found by the last call to BARCODE_Recognize. The returned code will be one of those listed above under Barcode Types.

BARCODE_Text

```
string BARCODE_Text(int n)
```

Return the text of the nth barcode found by the last call to BARCODE_Recognize. Barcodes found by BARCODE_Recognize are numbered from 0. If there is any problem of any kind, this function returns the empty string. In some programming languages this function is not available and you must use BARCODE_GetText (below).

BARCODE_GetText

```
BOOL BARCODE_GetText(int n, LPSTR Text)
```

Get the text of the nth barcode found by the last BARCODE_Recognize.
Please allow 64 characters in your text buffer. Use a smaller buffer only if you *know* that the barcode type is limited to shorter strings.

BARCODE_GetRect

```
BOOL BARCODE_GetRect(int n, double *L, double *T, double *R,
double *B)
```

Get the rectangle around the nth barcode found by the last BARCODE_Recognize, returning the top-left and bottom-right coordinates, in pixels, in the four parameters. (0,0) is the visual *top left corner* of the image. Returns TRUE(1) if successful, FALSE(0) otherwise. The only likely cause of a FALSE return would be an invalid value of n, or if you are in C or C++, a null pointer parameter.

Functions – Optical Character Recognition (OCR)

Introduction

Optical Character Recognition (*OCR)* is the industry term for the reading of text in an image by machine. You will also sometimes see it called *Intelligent Character Recognition (ICR)*. This is highly relevant to scanning because so much of what we scan contains text. Extracting the text on a scanned page can be useful for indexing documents, for searching them, and for automatic routing and processing.

EZTwain OCR is based on a multi-engine architecture with a specific set of supported engines in each release. You can enumerate the supported or available engines, select an engine, and use that engine to recognize text in scanned or loaded images.

EZTwain Pro currently supports only one OCR engine, the TOCR engine by Transym Computer Services Ltd (www.transym.com) *This engine is not provided by Atalasoft – it must be separately licensed and installed*. The TOCR engine was chosen because it was the engine most recommended by our customers for its speed and accuracy at plain text recognition. It outputs plain text – no font, font-size, style or other formatting information is provided.

Although this release of EZTwain Pro only supports one engine, we expect to support additional engines in the future. Please code defensively: Engine codes are constants and will never change, but the *default OCR engine* may change from release to release. At start-up, EZTwain Pro will select the available engine that we think will give the most satisfactory results for the greatest number of new customers. Keep in mind that in the future this may not be the Transym engine, or it could be a substantially different version of the Transym engine.

Using TOCR with EZTwain Pro is relatively simple:

1. Install the TOCR product according to Transym's directions.
2. In your application, start by selecting the TOCR engine using OCR_SelectEngine(EZOCR_ENGINE_TRANSYM) (Or the equivalent in your programming language.)
3. If that returns True (1) you may invoke other OCR services either directly using OCR_RecognizeDib, or indirectly using functions such as

OCR_IsAvailable

BOOL OCR_IsAvailable()

This function returns True (1) if any OCR services are available. This does *not* mean that any particular engine is available: Always check for the particular engine you prefer using OCR_IsEngineAvailable.

OCR_Version

int OCR_Version()

Returns the version number of the EZTwain Pro OCR subsystem, as the usual m.nn fraction multiplied by 100. So a version 1.25 OCR subsystem will return 125. Note that this is the version of *our* OCR subsystem, not the version of an OCR engine.

OCR_IsEngineAvailable
OCR_SelectEngine
OCR_SelectDefaultEngine
OCR_SelectedEngine
OCR_EngineName

```
BOOL OCR_IsEngineAvailable(int nEngine)
BOOL OCR_SelectEngine(int nEngine)
BOOL OCR_SelectDefaultEngine()
int OCR_SelectedEngine()
string OCR_EngineName(int nEngine)
```

These functions allow you to test for availability of a specific OCR engine, to select an engine, to see what the currently selected engine is, and to retrieve the human-readable name of any supported engine.

OCR Engine Codes

Symbol	Code	Description
EZOCR_ENGINE_NONE	0	'null' OCR engine - turns off OCR.
EZOCR_ENGINE_TRANSYM	1	TOCR engine by Transym Ltd.

Using OCR_EngineName, you can enumerate the supported OCR engines, to populate a listbox for example. Just call OCR_EngineName(i) with i = 0, 1, ... until it returns an empty string.

OCR_SetEngineKey

```
void OCR_SetEngineKey(string key)
```

Passes a registration/unlock key to the selected OCR engine.

For example, Transym Computer offers a *reseller* version of their TOCR engine. When you license this product, you receive a special version of TOCR, and a 16-digit registration number. Once the eller version of TOCR is installed on a computer, you can use it through EZTwain by passing in the registration number with a call like this:

OCR_SetEngineKey("0123-4567-89AB-CDEF")

OCR_SetEnginePath

```
void OCR_SetEnginePath(string path)
```

Sets the directory from which to load the OCR engine. You can set the path to "". This function does not check that the path exists, or that it has valid syntax.

By default the engine path is "", and EZTwain searches in the 'usual places' – according to the Windows DLL search sequence.

If the engine path is set to a non-empty string, EZTwain looks for the OCR engine *only* in the specified directory.

OCR_SetLineBreak

```
OCR_SetLineBreak(string sEOL)
```

Set the character sequence to use for line breaks in OCR'd text (as returned by OCR_Text and OCR_GetText).

- The default OCR line break is \n (LF or 0x0A)
- Other commonly used line breaks are \r (CR, 0x0D) or CRLF.
- Set this *before* doing OCR - it does not modify already recognized text.

OCR_RecognizeDib

```
int OCR_RecognizeDib(HDIB hdib)
```

Recognize text in the specified image, using the currently selected engine. The recognized text can be retrieved with OCR_Text or OCR_GetText, and the position information with OCR_GetCharPositions and OCR_GetCharSizes.

Return codes:

0	no error, but no text found.
n>0	n characters of text are available – including spaces and newlines.
-1	OCR services or selected engine not available.
-3	the image handle is null or invalid.
-5	there was an internal error or the OCR engine returned an error.

In case of an error, call TWAIN_ReportLastError, TWAIN_LastErrorCode, or similar functions for more details.

OCR_RecognizeDibZone

```
int OCR_RecognizeDibZone(HDIB hdib, int x, int y, int w, int h)
```

Recognize text in the specified rectangle (zone) of the specified image, using the currently selected engine. Otherwise identical to OCR_RecognizeDib.

Be sure you understand the parameters: (x,y,w,h) specify a rectangle w pixels wide, h pixels high, starting y pixels down from the top of the image and x pixels in from the left edge.

OCR_Text

```
string OCR_Text()
```

Returns the text recognized by the last call to OCR_RecognizeDib. If there is any problem, returns the empty string.

OCR_GetText

```
BOOL OCR_GetText(char *buffer, int buflen)
```

Retrieves the text recognized by the last call to OCR_RecognizeDib. It copies no more than buflen characters into buffer, including a terminating NUL (0 character) for those languages that require this. If successful, returns True (1), otherwise False (0).

OCR_TextLength

```
int OCR_TextLength()
```

Returns the number of characters in the stored OCR text. Does ***not*** include the terminating NUL, for those of you working in languages that care about that.

OCR_TextOrientation

```
Int OCR_TextOrientation()
```

Returns the orientation of the text found by the last OCR_RecognizeDib.
The value is the number of degrees clockwise that the input image was auto-rotated before OCR was performed.
Currently, the returned value is always a multiple of 90, so the only possible values are 0, 90, 180 and 270.

Example: If the original was turned 90 degrees clockwise before scanning, it will be auto-rotated 90 degrees *counter-clockwise* before OCR, so in that case the value of this function will be 270.

OCR_GetCharPositions
OCR_GetCharSizes

```
BOOL OCR_GetCharPositions(long x[], long y[])
BOOL OCR_GetCharSizes(long w[], long h[])
```

Retrieve the positions and sizes, respectively, of the characters recognized by the last call to OCR_RecognizeDib. Positions are in pixels relative to the *top left* corner of the processed image. Sizes are in pixels.

It is the caller's responsibility to ensure that x, y, w, and h are arrays of word-width integers, allocated large enough to hold N entries, where N is the character count returned by OCR_TextLength or the last call to OCR_RecognizeDib. Sorry we don't have example code yet.

OCR_ClearText

```
void OCR_ClearText()
```

Clear the text and other information stored by the last OCR recognition. After this call, OCR_TextLength will return 0, and OCR_Text and OCR_GetText will return the empty string.

OCR_WritePage

```
BOOL OCR_WritePage(HDIB hdib)
```

Recognize the text in the specified image, then write the image plus the (hidden) text to the currently open PDF output file. An available OCR engine must be selected. There must be a PDF file currently open for output, opened with TWAIN_BeginMultipageFile.

OCR_WriteTextToPDF

```
BOOL OCR_WriteTextToPDF()
```

Write the text from the last OCR to the next PDF page. The output text is retained until a page is written to a PDF file, then it is placed (invisibly) on that page.

Functions – Image Files

This section...

- describes the options and restrictions of each file format,
- explains how EZTwain decides which format to use when writing a file,
- describes which DLLs are required to support the various file formats,
- lists the functions that write images to files.

File Formats - Restrictions and Options

Format	Image Type(BPP)	Options
BMP		No options. No compression. Single page/image per file.
	BW(1), Palette(4,8), RGB(24)	These are standard BMP formats.
	Gray(8)	EZTwain can read and write this format, but some other programs may interpret a BMP of this format as a palette-color image with 256 colors (which all happen to be shades of gray...)
	Gray(16), RGB(48), CMY(24,48), CMYK(32)	EZTwain will read and write these non-standard formats in BMP, but few other programs will read them correctly.
TIFF		Many options. Most accomodating file format. EZTwain can append to an existing file. Single or multiple pages/images per file. See also: TIFF , page 97
	BW(1)	Default compression: CCITT Group 4 Fax, which does very well on scanned office documents. Other supported BW compressions are RLE (run-length encoding), CCITT Group 3 Fax, LZW, and 'Packbits'.
	Palette(4,8)	Default compression: None. Some palette images compress well with LZW.
	Gray(8), RGB(24), CMY(24), CMYK(32)	Default compression: None. JPEG compression is available, but creates compatibility problems with some older software.
	Gray(16), RGB(48),	These 'deep' images can be written (and read), but are always stored uncompressed.

	CMY(48)	
PDF		Highly flexible format. Supports single and multipage files. EZTwain can read its own PDF files, but *not most other PDFs*. EZTwain can append to its own PDF files, probably some others. See also: PDF , 100
	BW(1), Palette(4,8)	Always compressed with 'Flate' compression, which is a form of LZ compression.
	Gray(8), RGB(24), CMYK(24)	Always compressed with JPEG compression. Degree of compression controlled by TWAIN_SetJpegQuality. Note that CMY images are *not* supported.
JPEG	Gray(8), RGB(24), CMY(24)	Technically, EZTwain writes the JFIF file format, which is a non-progressive JPEG stream with some additional tags such as resolution. One page/image per file. Degree of compression controlled by TWAIN_SetJpegQuality. Not defined for BW or Palette images, nor for 'deep' images of > 8 bits/channel.
GIF	BW(1), Palette(4,8)	No options. Single image per file. Compression is always by LZW.
	RGB(24,48), CMY(24,48), CMYK(32)	These can be written to GIF format but they are *always converted to 8-bit palette color* before writing. This is suitable only for export, because it destroys so much of the information in the image.
DCX	BW(1)	No options. Rarely used format, commonly associated with facsimile applications. Multiple pages/images allowed in a file. Can be appended to, see: TWAIN_SetFileAppendFlag. Standard compression does well on documents.
PNG	BW(1), Palette(4,8), RGB(24)	No options. Single image per file. Standard compression is LZ, which does well on scanned printed or typed documents, poorly on images and photos.

File Format Codes (TWFF_* Codes)

Format Name	Code	Extension	Meaning
TWFF_TIFF	0	.tif, .tiff	Tagged Image File Format. Note: By default, Group4 Fax compression is used for 1-bit images, all others are uncompressed.
TWFF_BMP	2	.bmp	Windows Bitmap – uncompressed. Note: BMP support is built into EZTwain, so is always available.
TWFF_JFIF	4	.jpg, .jpeg	JPEG File Interchange Format 1.02
TWFF_PNG	7	.png	Portable Network Graphics
TWFF_DCX	97	.dcx	DCX - multipage PCX fax format.
TWFF_GIF	98	.gif	Graphics Interchange Format Note: TWFF_GIF is not a TWAIN constant, TWAIN does not recognize GIF. GIF support is only provided by EZTwain.
TWFF_PDF	99	.pdf	(Adobe) Portable Document Format Note: Same comment as for GIF above.

How EZTwain Chooses Output Format

If you use TWAIN_AcquireToFilename or DIB_WriteToFilename, the format of the output file is determined as follows:

If the specified filename ends in .BMP, .JPG, .JPEG, .TIF, .TIFF, .PNG, .GIF, .DCX or .PDF, then the file is saved in the corresponding format. Otherwise the current *Save Format* is used. The Save Format is set by TWAIN_SetSaveFormat and is initially BMP.

Similarly, TWAIN_AcquireMultipageFile will write PDF format if the filename ends with .PDF, TIFF format if the filename ends with .TIF, .TIFF or .MPT (Multi-Page Tiff), and DCX format if it sees a .DCX extension. If it does not recognize the file extension, it uses the current *Multipage Format* – which is set by TWAIN_SetMultipageFormat (p 35) and is initially TIFF.

File Format Support - Optional DLLs

The EZTwain main module (Eztwain4.dll) by itself can only read and write the BMP file format. To write the other file formats, the optional EZ* DLLs must be properly installed - See EZTwain Components, page 4.
Important: If you use either EZT4Tiff.dll or EZT4Pdf.dll, you must also install EZT4Jpeg.dll (whether or not you actually use JPEG compression.)

General file-writing settings

TWAIN_SetFileAppendFlag/TWAIN_GetFileAppendFlag

```
void TWAIN_SetFileAppendFlag(int nAppend)
int TWAIN_GetFileAppendFlag(void)
```

These functions set and query the *File Append Flag*. This flag controls what EZTwain does in the event of writing to a TIFF or DCX file which already exists. If the *File Append Flag* is non-zero and the program attempts to write to an existing TIFF or DCX file, EZTwain *appends* images to the existing file. Otherwise if the *File Append Flag* is 0 (the default case), writing a TIFF, DCX (or any other) file overwrites any previous contents of that file. Note: If there is no existing file, this flag is ignored.

TWAIN_SetJpegQuality / TWAIN_GetJpegQuality

```
void TWAIN_SetJpegQuality(int nQ)
int TWAIN_GetJpegQuality(void)
```

Sets the quality of JPEG compression throughout EZTwain, including any subsequently saved JPEG/JFIF file, or JPEG compressed image in PDF and TIFF format. You can use any value from 1 to 100, although I have never heard of anybody using a value below 40 in practice. This table lists some sample values for guidance. The sample compression gives the ratio of the uncompressed to the compressed JPEG file, for a 200 DPI RGB scan of a National Geographic magazine cover.

Quality	Description	Sample Compression
1	Lowest-quality, smallest files	150X
25	Low quality	40X
50	Moderate quality	20X
75	Good quality **[DEFAULT]**	12X
90	High quality	6X
100	Highest quality	2.5X

You cannot directly control the size of JPEG files – lower quality means smaller files, higher quality means larger, but the relationship is non-linear and depends on the content of the image being compressed.

Even at quality 100 JPEG is still a lossy compression - there will still be degradation of the image, although it is very unlikely to be detectable by the human eye. Nonetheless there are subtle mathematical changes in the image, and repeated compression and recompression even at quality level 100 can lead to cumulative (visible) image degradation.

PDF: By default, PDF uses JPEG compression for grayscale and RGB or CMYK color images. See PDF_SetCompression for more information.

TIFF: Subject to some warnings about compatibility, TIFF files can be written with JPEG compression (See TWAIN_SetTiffCompression.

Writing images to files

DIB_WriteToFilename/TWAIN_WriteToFilename

```
int DIB_WriteToFilename(HANDLE hdib, string pszFile)
int TWAIN_WriteToFilename(HANDLE hdib, string pszFile)
```

Writes an image to a file. If the file string ends with a recognized extension (BMP, JPG, JPEG, TIF, TIFF, PNG, GIF, DCX or PDF), then the file is written in the implied format. Otherwise, the file is written using the current save format: See TWAIN_SetSaveFormat. Normally if the output file exists it is overwritten, but TIFF, PDF and DCX files can be appended to: See TWAIN_SetFileAppendFlag.

hdib	DIB handle, as returned by TWAIN_Acquire
pszFile	filename string

If pszFile is NULL or points to a null string, the user is prompted for the filename *and format* with a standard Windows File Save dialog. The Save dialog will only offer formats that are available and valid for the given image.

Return values:

0	success
-1	user cancelled File Save dialog
-2	file open error (invalid path or name, or access denied)
-3	image is invalid, or cannot be written in this format.
-4	writing data failed, possibly output device is full

DIB_WriteArrayToFilename

int DIB_WriteArrayToFilename(HDIB ahdib[], int n, string File)

Write n images from array ahdib to the specified file.
If n is 1, this is exactly equivalent to calling DIB_WriteToFilename.
If n > 1, this is a shortcut for calling
TWAIN_BeginMultipageFile,
TWAIN_DibWritePage (for each image)
TWAIN_EndMultipageFile
...with appropriate error handling, of course.

Return values:

0	success
-1	user cancelled File Save dialog
-2	file open error (invalid path or name, or access denied)
-3	a) image is invalid (null or invalid DIB handle) b) support for the save format is not available (missing DLL?) c) DIB format incompatible with save format e.g. B&W to JPEG.
-4	writing data failed, possibly output device is full
-5	other unspecified internal error
-6	a multipage file is already open
-7	multipage support is not installed.

TWAIN_BeginMultipageFile

```
int TWAIN_BeginMultipageFile(string pszFile)
```

Create or open a multipage file. If pszFile is NULL(0) or points to an empty string, prompts the user for the file name, using a standard File Save dialog.

If the filename ends with .TIF, .TIFF, or .MPT, a TIFF file is started. If it ends with .DCX a DCX file is started. If it ends with .PDF a PDF file is started. Otherwise the file uses the current default multipage file format (see p 35), and if no extension is present, an appropriate extension for the format is added.

If the file already exists (and is writable) its content is deleted if the *File Append Flag* is 0, or it is appended to if the *File Append Flag* is non-zero. See TWAIN_SetFileAppendFlag.

Return values:

0	success.
-1	user was prompted for file and cancelled the File Save dialog.
-2	file open error (invalid path or name, or access denied)
-3	the required EZT4Tiff.dll, EZT4Dcx.dll or EZT4Pdf.dll failed to load.
-5	unexpected internal error
-6	multipage file already open.

TWAIN_DibWritePage

```
int TWAIN_DibWritePage(HANDLE hdib)
```

Append a page (image) to the currently open multipage file. This call will fail unless it follows a successful call to TWAIN_BeginMultipageFile.

Return values:

0	success.
-3	the required library (EZT4Tiff.dll, EZT4Dcx.dll, EZT4Pdf.dll) failed to load *or* invalid DIB or DIB handle, *or* image format not supported (e.g. 16-bit/pixel to PDF)
-4	Write error: Output device is full?
-5	unexpected internal error.
-6	multipage file not open.

TWAIN_EndMultipageFile

```
int TWAIN_EndMultipageFile(void)
```

Return values:

0	success.
-3	the required EZT4Tiff.dll, EZT4Dcx.dll or EZT4Pdf.dll failed to load.
-4	Write error: Output device is full?
-5	some internal error
-6	multipage file not open

TWAIN_IsMultipageFileOpen

```
BOOL TWAIN_IsMultipageFileOpen()
```

Returns True(1) if a multipage output file is open, False(0) otherwise.
Only one multipage output file can be open at a time (per process.)

TWAIN_MultipageCount

```
TWAIN_MultipageCount() => int
```

Return the number of images written to the most recently started multipage file. In other words, this returns a counter that is reset by BeginMultipageFile, and is incremented by DibWritePage. Can also be used during or after TWAIN_AcquireMultipageFile.

If you might be appending to a file and want to know the total page count of the file, see DIB_GetFilePageCount/TWAIN_PagesInFile 91.

TWAIN_SetOutputPageCount

```
void EZTAPI TWAIN_SetOutputPageCount(int nPages)
```

Tell EZTwain how many pages you are about to write to a file. This is OPTIONAL: The only effect is to add PageNumber tags to TIFF files. You can use nPages=0, which means "I don't know". See Faxing with TIFF: TIFF Class F and RFC 2301, p 99.

Loading images from files

DIB_LoadFromFilename

```
HANDLE DIB_LoadFromFilename(string pszFile)
```

Load an image from the specified file. If anything goes wrong the return value is NULL (0) - call TWAIN_LastErrorCode and related functions for details. If the file is multipage, normally the first image is loaded but see DIB_SelectPageToLoad. If pszFile is NULL or points to an empty string, the user is prompted to choose a file with a standard File Open dialog.

TWAIN_FormatOfFile

```
int TWAIN_FormatOfFile(string pszFile)
```

Return the format of the specified file. See File Format Codes (TWFF_* Codes) above. A return value < 0 means *unrecognized format*.

DIB_GetFilePageCount/TWAIN_PagesInFile

```
int DIB_GetFilePageCount(string pszFile)
int TWAIN_PagesInFile(string pszFile)
```

Return the number of pages in the specified file. The multipage formats supported are TIFF, PDF and DCX, all other recognized formats will return a page count of 1. A return value < 0 indicates an error: No such file, unreadable, unrecognized format, etc.

DIB_SelectPageToLoad

```
void DIB_SelectPageToLoad(int nPage)
```

For use when loading multipage files. Tells DIB_LoadFromFilename which page to load next, from a multipage file. Default is page 0 (first page in file). This value is reset to 0 after each call to DIB_LoadFromFilename. Example:

```
// Load all pages from file:
int N = DIB_GetFilePageCount(sFilename);
for (int i = 0; i < N; i++) {
     DIB_SelectPageToLoad(n);
     hdib[i] = DIB_LoadFromFilename(sFilename);
}
```

DIB_LoadPage

```
HDIB DIB_LoadPage(string pszFile, int nPage)
```

Short for DIB_SelectPageToLoad, DIB_LoadFromFilename.
Loads the specified page from the specified file. Page 0 is the first page in a file. Multiple pages are only supported in TIFF, PDF and DCX format, all other file formats have a single page (page 0).

Remember that EZTwain cannot generally read PDF files generated or modified by other software.

DIB_LoadArrayFromFilename

```
int DIB_LoadArrayFromFilename(HDIB ahdib[], int n,
                              string Filename)
```

Load up to n images as DIBs into an array, reading from the specified file.
If filename is null or the empty string, the user is prompted to select a file.

If the user is prompted and cancels, this function returns -10.
Otherwise if successful it returns the number of pages (images) loaded.
Otherwise it returns -1 and you should call TWAIN_ReportLastError, TWAIN_LastErrorCode,etc.

If this function returns < 0, the first n entries of the DIB array will be NULL (0).
If returns r >= 0, the first r entries of the DIB array will contain handles to DIBs representing the first r images in the file.
The remaining n-r entries in the DIB array will be NULL (0).

Remember to call DIB_Free on any DIB that you are no longer using.
As a convenience, there is a function DIB_FreeArray(array, n) which calls DIB_Free on the first n entries in an array of DIBs. It knows to ignore NULL(0) entries.

DIB_LoadPagesFromFilename

```
int DIB_LoadPagesFromFilename(HDIB ahdib[], int i, int n,
                              string  Filename)
```

Similar to DIB_LoadArray, this function loads n images from the specified file, starting with the ith image in the file (counting from 0 as the first image) into the ahdib array.

Very similar to DIB_LoadArrayFromFilename (above) except that it always loads n images or fails.

General file format inquiries

TWAIN_IsJpegAvailable
```
int TWAIN_IsJpegAvailable(void)
```

TWAIN_IsPngAvailable
```
int TWAIN_IsPngAvailable(void)
```

TWAIN_IsTiffAvailable
```
int TWAIN_IsTiffAvailable(void)
```

TWAIN_IsPdfAvailable
```
int TWAIN_IsPdfAvailable(void)
```

TWAIN_IsGifAvailable
```
int TWAIN_IsGifAvailable(void)
```

TWAIN_IsDcxAvailable
```
int TWAIN_IsDcxAvailable(void)
```

TWAIN_IsFormatAvailable
```
int TWAIN_IsFormatAvailable(int nFF)
```

Return TRUE (1) if the specified image file format is available i.e. the necessary EZ*.dll files can be found and loaded. Returns FALSE(0) if not.

TWAIN_FormatVersion
```
int TWAIN_FormatVersion(int nFF)
```

Returns the version number, times 100, of the module that implements the specified file format. For example, TWAIN_FormatVersion(TWFF_PDF) returns the version of the PDF module. A return value of 123 means version 1.23. If the format code is unrecognized or the file format module is not available, this function returns 0.

TWAIN_IsFileExtensionAvailable
```
int TWAIN_IsFileExtensionAvailable(string sExt)
```

This function takes a file-extension as a string and returns TRUE (1) if the corresponding file format support is available. It returns FALSE (0) if the format is not available (presumably because the required DLL or DLLs are not installed) *or* if it does not recognize the extension. Case does not matter, and a leading '.' is optional. Examples:
TWAIN_IsFileExtensionAvailable("tiff")
TWAIN_IsFileExtensionAvailable(".PNG")

TWAIN_FormatFromExtension

```
int TWAIN_FormatFromExtension(string sExt, int nFF)
```

Return the format implied by a file specification or extension. See File Format Codes (TWFF_* Codes) above. If the extension is not recognized, returns nFF. If you pass this function a filename it will parse it to find the extension. If the string contains no '.', it is assumed to be an exact extension string e.g. "tif". Case is ignored of course.

TWAIN_ExtensionFromFormat

```
string TWAIN_ExtensionFromFormat(int nFF, string sDefExt)
```

Return the default extension for the file format, *including leading* '.'. See File Format Codes (TWFF_* Codes) above. If nFF is not a valid format code, the string sDefExt is returned.

TWAIN_SetSaveFormat

```
int TWAIN_SetSaveFormat(int nFF)
```

Specifies the *default* file format for subsequent calls to DIB_WriteToFilename and TWAIN_AcquireToFilename. Displays a warning message if the format is not available: See TWAIN_IsFormatAvailable

This function is not normally needed: All functions that write an image file will *infer* the file format from the file extension. If your filenames include recognizable extensions like ".tif", you do not need to call TWAIN_SetSaveFormat.

Returns TRUE (1) if successful, FALSE (0) if format is invalid or not available.

TWAIN_GetSaveFormat

```
int TWAIN_GetSaveFormat()
```

Return the current *default save format*.

TWAIN_LastOutputFile

```
string TWAIN_LastOutputFile()
```

Return the name of the last file written by EZTwain.
Useful if you pass NULL or "" as a filename to DIB_WriteToFilename or TWAIN_AcquireToFilename, etc.
Not available in Visual Basic.

Functions – Image Files in Memory

Writing Images to Files in Memory

DIB_WriteToBuffer

```
int DIB_WriteToBuffer(HANDLE hdib, int nFormat, BYTE* pbuffer,
int nbMax)
```

Write the image into the buffer in the specified file format, not exceeding nbMax bytes. The return value is the actual size of the image, *which may be more or less than nbMax*. If the return value > nbMax, it means only part of the image was written, and the buffer needs to be bigger. If pBuffer is NULL or nbMax=0 this function simply returns the required buffer size in bytes.

A return value of <= 0 indicates an error, such as
The image is invalid (null or invalid DIB handle)
The file format is unrecognized, not supported, not installed, etc.
You can't save that kind of image in that format e.g. B&W image to JPEG format.
Insufficient memory for temporary data structures (or corrupted heap)
Other internal failure.

Call TWAIN_LastErrorCode and similar functions for more details.

DIB_WriteArrayToBuffer

```
int DIB_WriteArrayToBuffer(const HDIB ahdib[], int n, int
nFormat, BYTE* pBuffer, int nbMax)
```

A combination of DIB_WriteArrayToFilename and DIB_WriteToBuffer.
Writes n images in an array to a memory buffer in the specified file format.
See DIB_WriteToBuffer above for the meaning of pBuffer and nbMax.

Return value: See DIB_WriteToBuffer above.

Reading Images from Files in Memory

DIB_FormatOfBuffer

```
int DIB_FormatOfBuffer(const BYTE* pBuffer, int nBytes)
```

Like TWAIN_FormatOfFile, but examines a file stored in a memory buffer.

DIB_PageCountOfBuffer/DIB_BufferPageCount

```
int DIB_PageCountOfBuffer(const BYTE* pBuffer, int nBytes)
int DIB_BufferPageCount(const BYTE* pBuffer, int nBytes)
```

Return the number of pages (images) in a file stored in a memory buffer.

DIB_LoadFromBuffer

```
HANDLE DIB_LoadFromBuffer(const BYTE* pBuffer, int nBytes)
```

Load an image from a buffer in memory containing data formatted as an image file. For multipage files, if DIB_SelectPageToLoad was called first the designated page will be loaded, otherwise the first image in the file is loaded.
pBuffer is the address of the buffer (memory block) holding the file to read.
nBytes is the number of bytes of data in buffer.

Error handling is same as for DIB_LoadFromFilename .

DIB_LoadPageFromBuffer

```
HDIB DIB_LoadPageFromBuffer(const BYTE* pBuffer, int nBytes,
                            int nPage)
```

Load the specified page from a buffer - the buffer must contain data in a supported image file format. If the image format is one that can hold only one image, the page number is ignored.
pBuffer is the address of the buffer (memory block) holding the file to read.
nBytes is the number of bytes of data in buffer.
nPage is the index of the page (image) to read, counting from 0.

DIB_LoadArrayFromBuffer

```
int DIB_LoadArrayFromBuffer(HDIB ahdib[], int nMax,
                            const BYTE* pBuffer, int nBytes)
```

Load up to nMax images as DIBs into an array, reading from a file in memory.
pBuffer is the address of the buffer (memory block) holding the file to read.
nBytes is the number of bytes of data in the buffer.

Returns the number of images loaded if successful, otherwise it returns -1 and you should call TWAIN_ReportLastError, TWAIN_LastErrorCode, or similar.

Make sure you eventually call DIB_Free, or DIB_FreeArray to free unused DIBs.

Functions - TIFF Specific

TWAIN_SetTiffCompression/TWAIN_GetTiffCompression

```
int TWAIN_SetTiffCompression(int nPT, int nComp)
int TWAIN_GetTiffCompression(int nPT)
```

Set or get the compression mode to use when writing TIFF files. You must specify the *pixel type* (image type) to which this compression applies (See p. 44 for the pixel type codes.) If you are not familiar with the properties of these compression algorithms, try Google – or contact Technical Support.

TIFF Compression Modes

Constant Name	Value	Compression Algorithm	Applies to
TIFF_COMP_NONE	1	No compression	*all*
TIFF_COMP_CCITTRLE	2	CCITT modified Huffman RLE	BW
TIFF_COMP_CCITTFAX3	3	CCITT Group 3 fax encoding[1]	BW
TIFF_COMP_CCITTFAX4	4	CCITT Group 4 fax encoding	BW
TIFF_COMP_LZW	5	LZW[2,3]	*all*
TIFF_COMP_JPEG	7	JPEG-in-TIFF[4]	RGB, GRAY

The default compression for 1-bit BW is TIFF_COMP_CCITTFAX4, and for all other image types is TIFF_COMP_NONE.
Note 1: Enables *TIFF Class F* - see comments on page 99.
Note 2:The Unisys patent on LZW compression has *expired.*
Note 3: LZW compression works poorly on almost all scans and camera images.
Note 4: JPEG-in-TIFF has compatibility issues: EZTwain writes *revised* TIFF 6 JPEG.

TWAIN_SetTiffStripSize/TWAIN_GetTiffStripSize

```
void TWAIN_SetTiffStripSize(int nBytes)
int TWAIN_GetTiffStripSize(void)
```

By default, images in TIFF files are stored in horizontal *strips* with a default size of 32768 bytes (roughly). A few *nonconforming* TIFF Readers cannot handle images with more than one strip, or images with large strips. Use these functions to work around this. Setting the TIFF strip size to –1 causes all images to be written using 1 strip.

TWAIN_SetTiffImageDescription
TWAIN_SetTiffDocumentName

```
void TWAIN_SetTiffImageDescription(string pszText)
void TWAIN_SetTiffDocumentName(string pszText)
```

These functions set the value of two standard TIFF tags, ImageDescription and DocumentName. These apply *only* to the next TIFF file written by EZTwain, and when EZTwain finishes writing a TIFF file, it forgets (clears) these values.

TWAIN_SetTiffTagShort
TWAIN_SetTiffTagLong
TWAIN_SetTiffTagDouble
TWAIN_SetTiffTagString
TWAIN_SetTiffTagRational
TWAIN_SetTiffTagRationalArray
TWAIN_SetTiffTagBytes
TWAIN_ResetTiffTags

```
BOOL TWAIN_SetTiffTagShort(int tag, short sValue)
BOOL TWAIN_SetTiffTagLong(int tag, long nValue)
BOOL TWAIN_SetTiffTagString(int tag, const char* pzText)
BOOL TWAIN_SetTiffTagDouble(int tag, double dValue)
BOOL TWAIN_SetTiffTagRational(int tag, double dValue)
BOOL TWAIN_SetTiffTagRationalArray(int tag, double da[], int n)
BOOL TWAIN_SetTiffTagBytes(int tag, const BYTE* pdata, int nLen)
void TWAIN_ResetTiffTags(void)
```

The tag values you set with these functions will be included in *each image subsequently written to TIFF* until you call TWAIN_ResetTiffTags.

The TIFF standard is available through this website:
http://www.remotesensing.org/libtiff/document.html

The specific data formats needed by each tag are documented here:
http://www.remotesensing.org/libtiff/man/TIFFSetField.3tiff.html

We recommend you use the Set function that corresponds to the TIFF data type of the tag, although TWAIN_SetTiffTagDouble will correctly set any standard RATIONAL, SRATIONAL, FLOAT or DOUBLE tag, and TWAIN_SetTiffTagLong will set any standard SHORT, LONG, or SLONG tag.

Custom and private TIFF tags: Please reference the TIFF standard for more details on private and custom tags. The SetTiffTag functions can be used to set such tags - However for such tags you *must* use the function of the correct type: No conversion will be performed.

Example of setting TIFF tags:

```
// Save hdib to TIFF, with artist and 'bad fax lines':
#define TIFFTAG_ARTIST          315
#define TIFFTAG_BADFAXLINES     326
TWAIN_SetTiffTagString(TIFFTAG_ARTIST, "ABBA")
TWAIN_SetTiffTagLong(TIFFTAG_BADFAXLINES, 0)
 DIB_WriteToFilename(hdib, "c:\DancingQueen.tif")
TWAIN_ResetTiffTags()
```

TWAIN_GetTiffTagAscii / TWAIN_TiffTagAscii

```
BOOL TWAIN_GetTiffTagAscii(string file, int page, int tag, int
len, char buffer)
string TWAIN_TiffTagAscii(string file, int page, int tag)
```

Read the value of an ASCII-string-valued TIFF tag from the specified page of the specified TIFF file.

TWAIN_TiffTagAscii returns the tag value as a string, returning the empty string if anything goes wrong.

TWAIN_GetTiffTagAscii copies the string into buffer, which has room for at least len characters. Usually the buffer variable mst be allocated or resized before you call this function, to reserve the space. Returns True(1) if successful, False(0) otherwise.

Faxing with TIFF: TIFF Class F and RFC 2301

There are several variations of TIFF specialized for facsimile (fax) applications.

TIFF Class F is a variant of TIFF for storing faxes, which according to one source "has been in common usage for many years" - as of 1997. It is not an official standard. See for example:
http://palimpsest.stanford.edu/bytopic/imaging/std/tiff-f.html

IETF RFC 2301 - "File Format for Internet Fax" is a draft Internet standard:
http://www.ietf.org/rfc/rfc2301.txt

EZTwain Pro can write TIFF Class F files: You will need to set a few tags 'by hand', as demonstrated by this code fragment which writes out a DIB. Note that the image needs to be B&W (1 bit/pixel), have a width of 1728, 2048, or 2482 pixels, and have a resolution of 204 DPI horizontal and either 98 or 196 DPI vertical.

```
HDIB hdib = DIB_Allocate(1, 1728, 1056);
DIB_SetResolution(hdib, 200, 96);
// Select Group3 Fax compression for B&W TIFF:
TWAIN_SetTiffCompression(TWPT_BW, TIFF_COMP_CCITTFAX3);
// Set the FillOrder tag to 'Least Significant Bit First'
#define FILLORDER_LSB2MSB 2
TWAIN_SetTiffTagLong(TIFFTAG_FILLORDER, FILLORDER_LSB2MSB);
// Set the option for 'byte aligned EOLs'
#define GROUP3OPT_FILLBITS 4
TWAIN_SetTiffTagLong(TIFFTAG_GROUP3OPTIONS, GROUP3OPT_FILLBITS);
// Don't break images into strips:
TWAIN_SetTiffStripSize(-1);
// Tell EZTwain how many pages will be in the file:
TWAIN_SetOutputPageCount(1);
DIB_WriteToFilename(hdib, "classF.tif");
```

Functions - PDF Specific

PDF_SetTitle
PDF_SetAuthor
PDF_SetSubject
PDF_SetKeywords
PDF_SetCreator
PDF_SetProducer

```
int PDF_SetTitle(string pzText)
int PDF_SetAuthor(string pzText)
int PDF_SetSubject(string pzText)
int PDF_SetKeywords(string pzText)
int PDF_SetCreator(string pzText)
int PDF_SetProducer(string pzText)
```

These functions set the value of standard keys in the document information dictionary of the next PDF file. These apply *only* to the next PDF file written by EZTwain, and when EZTwain finishes writing a PDF file, it forgets (clears) these values.

EZTwain defaults both 'Creator' and 'Producer' to "EZTwain Pro n.nnbn using EZPdf n.nn"

PDF_DocumentProperty

```
String PDF_DocumentProperty(string Filename, string Property)
```

Read the specified property (from the document information dictionary) of the specified PDF file and return it. These are the same document properties written to PDF files by the functions PDF_SetTitle, PDF_SetAuthor, and so on.

In languages with 'char pointers' the return value is a pointer to an ephemeral internal buffer that should be copied immediately.

Valid property names are:

Title Author Subject Keywords Creator Producer

Case is significant, use the exact property strings above.

In case of any failure or error, the return value will be the empty string ("") and an error will be recorded which as usual can be displayed with TWAIN_ReportLastError, examined with TWAIN_LastErrorCode, and so on.

PDF_GetDocumentProperty

```
int PDF_GetDocumentProperty(
      string Filename, string Property,
      char *buffer, int buflen)
```

Like PDF_DocumentProperty, but the property's value string is copied into the buffer.
If buflen = 0, the buffer is not touched – in fact buffer can be NULL.
If buflen > 0, up to buflen-1 bytes of the property value string are copied into the provided buffer, followed by a NUL(0) byte. Note: The return value is the true full length of the property value string found in the file, regardless of the value of buflen.

PDF_SetCompression

```
BOOL PDF_SetCompression(int pt, int comp)
int PDF_GetCompression(int pt)
```

Set or query the compression algorithm to use when images of the specified pixel type are written to PDF.
Special cases:

pt=-1 means *all applicable pixel types.*
comp=-1 means *default compression for the pixel type*.

Thus PDF_SetCompression(-1,-1) resets the compression for all pixel types to the default.

For pixel types codes and definitions, see *EZTwain Pixel Types*, page 44.

PDF Compression Choices

Constant Name	Value	Compression Algorithm	Applies to
COMPRESSION_DEFAULT	-1	Default for pixel type.	*all*
COMPRESSION_NONE	1	No compression	*all*
COMPRESSION_FLATE	5	Flate – a 'zip' compression	*all*
COMPRESSION_JPEG	7	JPEG, also called DCT.	gray, RGB

PDF_SelectPageSize

```
BOOL PDF_SelectPageSize(int nPaper)
```

Designates the page size for subsequent PDF output. See *Standard Paper Sizes*, page 136. By default, the PDF output page size is PAPER_NONE which means when an image is written to PDF, it is placed on a page that is the same size as the image. Images of unknown size (0 DPI) or abnormally small DPI, are arbitrarily reinterpreted as being 100 DPI. If you select a standard paper size, each image subsequently written to PDF is placed on a page of that size, and the image is reduced proportionally, if necessary, to fit on the page.

You can return to the default setting at any time by calling PDF_SelectPageSize with an argument of PAPER_NONE.

PDF_SelectedPageSize

```
int PDF_SelectedPageSize()
```

Returns the current page size for PDF output. See PDF_SelectPageSize above.

PDF_DrawText

```
PDF_DrawText(double x, double y, string text)
```

Draw the specified text into the next PDF page that is written, at coordinates (x,y) on the page. Normally this function is used to draw text on a page that consists of a single image, such as a scanned page. In this case, the coordinates x and y are in *pixels* relative to the top-left of the image.

PDF_SetTextVisible

```
PDF_SetTextVisible(BOOL bVisible)
```

Sets the visibility of the text drawn by subsequent PDF_DrawText calls.

PDF_DrawInvisibleText

```
PDF_DrawInvisibleText(double x, double y, string text)
```

Like PDF_DrawText, but always draws the text in invisible mode, i.e. as *hidden text*. Does not change the text visibility mode.

PDF Encryption / PDF Passwords

Starting with release 3.30, EZTwain Pro can write and read back encrypted, password-protected PDF files, in accordance with the PDF 1.4 specification.

A PDF file can be protected with a *user password*, an *owner password*, or both. In theory, a user must present one of the passwords to gain access to the contents of the file. Quoting from *PDF Reference, Third Edition* (PDF 1.4):

> A PDF document can be *encrypted (PDF 1.1)* to protect its contents from unauthorized access. Encryption applies to all strings and streams in the document's PDF file, but not to other object types such as integers and boolean values, which are used primarily to convey information about the document's structure rather than its content. Leaving these values unencrypted allows random access to the objects within a document, while encrypting the strings and streams protects the document's substantive contents.

The encryption used by this feature, a 40-bit RC4 encryption, while not *weak*, is not particularly strong by current standards. For more information and technical details, please refer to the PDF Reference, or to the many sources on the web.

The expected behavior of PDF reading programs (such as Acrobat Reader or Foxit Reader) is described this way in the PDF Reference:

> If a user attempts to open an encrypted document that has a user password, the viewer application should prompt for a password. Correctly supplying either password allows the user to open the document, decrypt it, and display it on the screen. If the document does not have a user password, no password is requested; the viewer application can simply open, decrypt, and display the document. Whether additional operations are allowed on a decrypted document depends on which password (if any) was supplied when the document was opened and on any access restrictions that were specified when the document was created:
>
> • Opening the document with the correct owner password (assuming it is not the same as the user password) allows full (owner) access to the document. This unlimited access includes the ability to change the document's passwords and access permissions.
>
> • Opening the document with the correct user password (or opening a document that does not have a user password) allows additional operations to be performed according to the user access permissions specified in the document's encryption dictionary.

When EZTwain functions are called to read an image or other content from an encrypted PDF file, they follow the guidelines given above with one difference: EZTwain first checks to see if the application has supplied a password using the PDF_SetOpenPassword function below. If so, that password is used exactly as if the user had entered it at a password prompt. Otherwise, *if the PDF has a user password*, the user is prompted for a password. If the supplied password matches either the user or owner password of the document, the document is opened and the

requested content is retrieved. EZTwain Pro does not and *cannot* enforce any access restrictions.

Encryption and Appending to an Existing PDF

When appending to an existing PDF, EZTwain requires that the encryption (if any) of the new data being written matches that of the existing file.

A. If the existing file is not encrypted, the appended new data may not be encrypted.

B. If the existing file is encrypted, then there are two sub-cases:

1. If the application has set UserPassword or OwnerPassword, the password or passwords must validate against the existing PDF. If they do, the new appended material is encrypted to match the existing PDF contents.

2. If no encryption password has been set by the application, the user is prompted for a password (as if the existing PDF was being opened for input). If that password validates against the existing PDF, then the new appended material is encrypted to match the existing PDF contents.

If any password supplied by the application or the user does not validate against the existing PDF, the attempted append is aborted without any effect on the file, and the calling function returns an error, probably EZTEC_PDF_PASSWORD.

PDF_SetOpenPassword

```
PDF_SetOpenPassword(string sPass)
```

Sets the password to be used to open encrypted PDF files. When EZTwain is asked to read content (usually an image) from an encrypted PDF, if the open password has been set with this function, that password is used to attempt to open the PDF, as if the user had entered it at a password prompt. If no password has been defined using this function, then the PDF read routines follow the guidelines quoted above from the PDF Reference.

PDF_SetUserPassword

```
PDF_SetUserPassword(string sPass)
```

Specify the user password for the next PDF file to be written. Setting a non-empty password for a PDF file causes that PDF to be encrypted using the standard encryption of PDF 1.4 as discussed above. When EZTwain completes writing a PDF file, this password is cleared to the empty string.

PDF_SetOwnerPassword

```
PDF_SetOwnerPassword(string sPass)
```

Define an owner password for the next output PDF file. Setting a non-empty password for a PDF file causes that file to be encrypted.

When a PDF file is completed and closed, the owner password is cleared.

PDF_SetPermissions / PDF_GetPermissions

```
PDF_SetPermissions(long nPermMask)
long PDF_GetPermissions()
```

Set or Get the access permissions mask to be written into the next output PDF file. This mask asks PDF viewer programs to restrict certain activities by the user, beyond simply opening and viewing the file.

- Permissions are only written if you set a user or owner password.
- It is a *permission* mask – 1 bits mean 'allow', 0 bits mean 'prevent'.
- Acrobat honors these restrictions, but other PDF readers may not.
- The permission mask you set applies only to the next PDF file written.
- The default permissions mask is 'allow everything' (-1)
- Setting a mask of 0 means 'prevent everything preventable'.

You can use bitwise operations, or +/- to combine these constants, for example, to disallow copying text and graphics from the file:'

```
PDF_SetPermissions(PDF_PERMIT_ALL - PDF_PERMIT_COPY)
```

Bit	Value	Named Constant	Operation (permitted if bit is set)
1	1		*(unused)*
2	2		*(unused)*
3	4	PDF_PERMIT_PRINT	printing the document
4	8	PDF_PERMIT_MODIFY	making changes, other than notes & form fields
5	16	PDF_PERMIT_COPY	copying or extracting content
6	32	PDF_PERMIT_ANNOTS	adding or changing comments or form fields
all	-1	PDF_PERMIT_ALL	All of the above

PDF/A – ISO 19005

PDF/A is a file format, a proper subset of Adobe PDF 1.4, defined by international standard ISO 19005-1 in 2005. It was created to facilitate the long-term storage of digital documents. Quoting from the Introduction to that standard:

> "The primary purpose of this part of ISO 19005 is to define a file format based on PDF, known as PDF/A, which provides a mechanism for representing electronic documents in a manner that preserves their visual appearance over time, independent of the tools and systems used for creating, storing or rendering the files."

For an overview and answers to frequently asked questions about PDF/A, see: http://www.aiim.org/documents/standards/PDF-A/19005-1_FAQ.pdf

As a proper subset of PDF 1.4, PDF/A files should be readable by any PDF reader which conforms to PDF 1.4 or higher.

PDF/A-1 files must include:

- Embedded fonts
- Device-independent color
- XMP metadata

PDF/A-1 files may not include:

- Encryption
- LZW Compression
- Embedded files
- External content references
- PDF Transparency
- Multi-media
- JavaScript

The published PDF/A-1 standard may be purchased directly from ISO or from national standards bodies around the world, such as ANSI (the American National Standards Institute).

PDF_SetPDFACompliance
PDF_GetPDFACompliance

```
BOOL PDF_SetPDFACompliance(int nLevel)
int PDF_GetPDFACompliance()
```

Set or get the PDF/A compliance level.

Level 0 is 'no particular compliance'
Level 1 tells EZTwain to write PDF files that conform to *ISO 19005-1 Level B*.

Functions – File Uploading

Overview

EZTwain Pro provides some basic services for uploading one or more images to a webserver, using the HTTP 'POST' command. The upload functions emulate the HTTP handshake produced by a web browser when a user submits a form containing a file-selection control. However using the EZTwain UPLOAD feature, no browser is required and there is no user interaction.

This feature of EZTwain Pro is optional - it is implemented in the EZT4Curl.dll, which you may omit from your software configuration if you do not need uploading services. As its name suggests, EZT4Curl.dll is a custom build of the highly respected open-source libcURL library by Daniel Stenberg.

UPLOAD_IsAvailable

```
BOOL UPLOAD_IsAvailable()
```

Returns True(1) if uploading services are available, False(0) otherwise. Currently this means that the EZT4Curl.dll has been found and loaded successfully, see the overview comments.

UPLOAD_Version

```
int UPLOAD_Version()
```

Returns the version number of the upload services module, EZT4Curl.dll, *as an integer*: major version * 100 + minor version. For example at the time of this writing, this function returns 715 which signifies EZT4Curl version 7.15.

UPLOAD_MaxFiles

```
int UPLOAD_MaxFiles()
```

Returns the maximum number of files that can be uploaded in one upload operation. At the time of this writing, this function returns 999.

UPLOAD_AddFormField

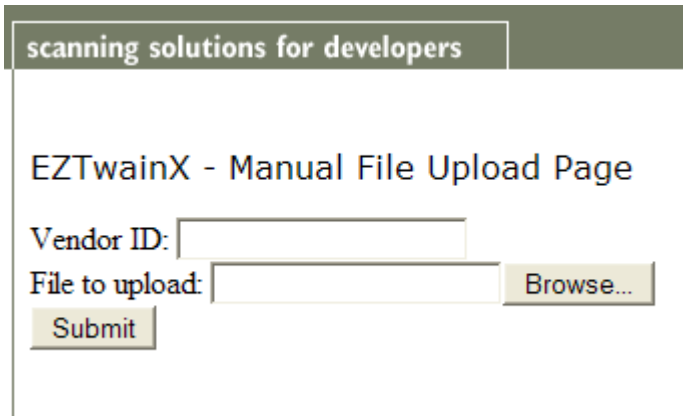
```
BOOL UPLOAD_AddFormField(string name, string value)
```

Set a form field to a value for the next upload. The name of the field must be expected by the page/script you upload to. All fields set with this function are discarded and forgotten after the upload that uses them.

This function returns True if successful, False otherwise. It can fail if more than 32 fields are defined prior to an upload, or if either argument is NULL. A successful return means only that the field was recorded, not that it was sent to or received by the server.

For example, suppose you have been uploading scanned documents to your server using a web form like this:

```
  <form name="form1" method="post" action="upload.php"
enctype="multipart/form-data">
    Vendor ID: <input type="text" name="vendor id"><br>
    File to upload:
    <input type="file" name="file"><br>
    <input type="submit" name="Submit" value="Submit">
  </form>
```



You might replace this form with an automatic upload of a scanned document with code similar to this:

```
    UPLOAD_AddFormField("vendor id", "1290331")
    UPLOAD_DibToURL(hdib, "http://eztwain.com/eztx/upload.php",
"file.pdf", "file")
```

UPLOAD_AddHeader

```
BOOL UPLOAD_AddHeader(string header)
```

Add the specified line to the HTTP header of the next upload. This can be used, for example, to send a cookie or a pragma to the server. This allows you to tinker with the headers sent by EZTwain Pro.

UPLOAD_AddCookie

```
BOOL UPLOAD_AddCookie(string cookie)
```

Add a cookie line to the next HTTP upload.
Often used to provide session id's, for example:
UPLOAD_AddCookie("ASP.NET_SessionID=" & strSessionID)
or
UPLOAD_AddCookie("JSESSIONID=" & strSessionID)

UPLOAD_EnableProgressBar
UPLOAD_IsEnabledProgressBar

```
UPLOAD_EnableProgressBar(BOOL bEnable)
BOOL UPLOAD_IsEnabledProgressBar()
```

Enable or disable – that is, show or hide - the progress-bar that appears during uploads. The default state of this setting is *enabled* (True)

UPLOAD_DibToURL
UPLOAD_DibsToURL
UPLOAD_DibsSeparatelyToURL
UPLOAD_FilesToURL

```
int UPLOAD_DibToURL(HDIB hdib, string url, string filename,
string field)
int UPLOAD_DibsToURL(HDIB ahdib[], int n, string url, string
filename, string field)
int UPLOAD_DibsSeparatelyToURL(HDIB ahdib[], int n, string url,
string filename, string field)
int UPLOAD_FilesToURL(string files, string url, string field)
```

Upload an image, a set of images, or a set of files to a script on a server, emulating a form being submitted from a browser via HTTP-POST.

Parameters

hdib	handle to image to upload
ahdib	address or reference to array of image handles
n	number of images to take from ahdib
url	the receiving script as a URL. For example: `http://www.eztwain.com/upload.php`
filename	the (pretended) name of the uploaded file. This is not the name of an actual file! The images to be uploaded are collected into a temporary file, and POSTed to the server: The server is told that it is receiving a file of this name. The extension on this filename determines the format of the uploaded file: .tif for TIFF format, .jpg for JPEG format, and so on.
files	A string containing one or more filenames, separated by semicolons (;) or vertical bars (|) e.g. "c:\file1.jpg;c:\file2.tif"
field	the name of the file-upload field on the form. Some scripts require a specific field name. When multiple files are being sent to the server, the value of field is modified by appending "1", "2", etc. to it.

Operation

All of these functions have in common that they emulate a web browser submitting to a server a multipart form with one or more files attached. The UPLOAD_Dib... functions do not actually read or create the named files – they just send the data to the server *as if* such a file was being uploaded. On the other hand, UPLOAD_FilesToURL expects to find the specified file or files on the local disk, and it uploads their contents and sends along their names.

A call to UPLOAD_DibToURL(hdib, *http:server/script*, *filename*, *field*) looks to the server script as if the user had browsed to a page on *server*, viewed the following form, selected a local file named *filename* and submitted the form:

```
<form name="form1" method="post" action="script"
enctype="multipart/form-data">
Upload this file:
    <input type="file" name="field">
    <input type="submit" name="Submit" value="Submit">
</form>
```

UPLOAD_AddFormField can be used to insert additional fields in the upload form, UPLOAD_AddHeader can be used to 'tweak' the HTTP header of the upload, and UPLOAD_EnableProgressBar can be used to hide or suppress the progress bar that EZTwain Pro normally displays during an upload.

Return values

0	success (transaction completed) **Note:** A success return means only that the data was sent to the server and a response was received, not that the receiving script necessarily accepted the submitted file. See DIB_UploadResponse below.
-1	user cancelled File Save dialog (should never happen!)
-2	could not write temp file - access denied, volume protected, etc.
-3	a) image is invalid (null or invalid DIB handle) b) The DLL(s) needed to save that format failed to load c) DIB format incompatible with save format e.g. uploading a B&W image as JPEG. d) filename extension isn't one EZTwain recognizes.
-4	writing data failed, maybe the disk with the temp folder is full?
-5	other unspecified internal error
-1xx	libcurl (the library EZTwain uses) returned error code xx For example:
-106	Could not resolve host
-107	Couldn't connect
-126	(UPLOAD_FilesToURL only) The specified files could not be opened and read.
-155	Connection was aborted.

Server Response

These functions deal with the text returned by the server in response to an UPLOAD operation such as UPLOAD_DibToURL. EZTwain collects and stores the text returned by the server in response to the last upload, up to an implementation-defined limit, *currently* around 12000 bytes.

UPLOAD_ResponseLength

```
int UPLOAD_ResponseLength()
```

UPLOAD_ResponseLength returns the number of characters returned by the server to the last Upload request, up to the maximum EZTwain can store.

UPLOAD_ClearResponse

```
void UPLOAD_ClearResponse()
```

UPLOAD_ClearResponse clears the stored response text. You usually don't need to call UPLOAD_ClearResponse, all the UPLOAD functions call it when they start.

UPLOAD_Response

```
string UPLOAD_Response()
```

UPLOAD_Response returns the text received from the server/script, in response to the last upload. Assume that this string can be any length and code defensively: Use UPLOAD_ResponseLength if necessary to preallocate storage. This will be the empty string before any upload, and after an upload that returns a negative status code. If you language permits, we recommend using UPLOAD_Response rather than UPLOAD_GetResponse.

UPLOAD_GetResponse

```
void UPLOAD_GetResponse(string s)
```

UPLOAD_GetResponse copies the last upload server response into a string parameter. T*his text is never more than 1024 characters long* - If you are using UPLOAD_GetResponse, please pre-allocate your string variable accordingly.

Functions – Image Viewing

TWAIN_ViewFile

```
int TWAIN_ViewFile(string pszFile)
```

Opens an image viewer window and displays the specified image file in it. The window can be resized by the user. If the file contains multiple pages/images, controls are displayed for stepping between the pages. The filename is displayed as the title/caption of the window. By default, the dialog is *modal* with an [OK] button.

The operation of this function can be modified using TWAIN_SetViewOption below.

Return values:

-1	error creating the window or opening/reading the file
0	user cancelled the window (by clicking the close box)
1	user clicked the OK button.

Caution: EZTwain cannot generally read PDF files generated or modified by other software.

DIB_View

```
int DIB_View(HDIB hdib, string pzTitle, HWND hwndParent)
```

Display the given image in a viewer window with the given title.
If hdib is 0 (NULL), the viewer window still opens but no image is displayed.
hwndParent is the window handle of the parent window - if you use 0 (NULL) for this parameter, EZTwain uses the active window of the application if there is one, or no parent window.
By default, the window contains only an [OK] button, the style of the window is a resizable dialog box, the dialog is *modal*, and this function does not return until the user closes the dialog or clicks the [OK] button.

The operation of this function can be modified using DIB_SetViewOption below.

DIB_SetViewImage

```
BOOL DIB_SetViewImage(HDIB hdib)
```

If the image viewer is open, this displays the specified image in the viewer window. To use this function, first call DIB_SetViewOption("modeless", "true") and then DIB_View(0, "<title>", 0) This opens the image viewer window with no current image. Then you can call DIB_SetViewImage repeatedly to display images, and DIB_ViewClose when you are done.

DIB_IsViewOpen/TWAIN_IsViewOpen

BOOL DIB_IsViewOpen()

Returns TRUE(1) if the viewer window is open, FALSE(0) otherwise. Normally this is only possible if the viewer is operating as a *modeless* dialog - set DIB_SetViewOption.

DIB_ViewClose/TWAIN_ViewClose

```
BOOL DIB_ViewClose()
```

Close the image viewer window if it is open. If it is not open, do nothing.

DIB_SetViewOption/TWAIN_SetViewOption

```
BOOL DIB_SetViewOption(string option, string value)
```

Set the value of an option related to TWAIN_ViewFile or DIB_View. Supported options and values are listed below.

Image Viewer Options[2]

Option	Value	Effect
"modal"	"true"[1,3]	Operate the viewer window as a *modal dialog.* Do not return from DIB_View or TWAIN_ViewFile until the user closes the viewer window.
"modal"	"false"[4]	Operate the viewer window as a *modeless dialog.* When DIB_View or TWAIN_ViewFile is called, display the image in the viewer window and return immediately, leaving the viewer window open.
"modeless"	"true"[3]	same as "modal","false" - viewer is *modeless.*
"modeless"	"false"[1,4]	same as "modal","true" - viewer is *modal.*
"position"	"x,y" *or* "x,y,w,h"	Set the position of the viewer window. X,y,w and h can be integers which are interpred as pixels. If a number is followed by a percent-sign (%) it is interpreted as that percent of the available screen width or height. w and h are optional.
"x" or "left"	"n" or "n%"	Set the left (x) coordinate of the viewer. As with position, n means pixels from the left side of the work area, n% means n% of the screen width.
"y" or "top"	"n" or "n%"	Set the top (y) coordinate of the viewer.
"width"	"n" or "n%"	Set the width of the viewer window.
"height"	"n" or "n%"	Set the height of the viewer window.
"size"	"w,h"	Set the width and height of the viewer window.
"visible"	"true"[1,3]	Show the viewer dialog, when it is modeless.
"visible"	"false"[4]	Hide the viewer dialog, when it is modeless.
"reset"	*(ignored)*	reset to default value all options that have one.
"title"	"any string"	Set the title bar text of the viewer window.
"ok.visible"	"true"[1,3]	Show the [OK] button in the viewer.
"ok.visible"	"false"[4]	Don't show the [OK] button.
"cancel.visible"	"true"[3]	Show the [Cancel] button in the viewer.
"cancel.visible"	"false"[1,4]	Don't show the [Cancel] button.
"print.visible"	"true"[3]	Show the [Print...] button in the viewer
"print.visible"	"false"[1,4]	Don't show the [Print...] button.

1: Default value.
2. We show all options and values with quotes because they are strings. Your language may use another way of quoting strings.
3: In place of "true" you may use: "1", "yes", "vrai", "oui", "si", or "ja".
4: In place of "false" you may use: "0", "no", "faux", "non", or "nein".

Functions – Error Handling & Logging

TWAIN_SuppressErrorMessages

```
int TWAIN_SuppressErrorMessages (int nSuppress)
```

Enable or disable EZTwain error messages to the user.
Returns the previous state of the flag.
When nSuppress = 0, error messages are displayed.
When nSuppress <> 0, error messages are suppressed.
By default, error messages are displayed.
Note that EZTwain cannot prevent message boxes displayed by TWAIN or DSs.

TWAIN_ReportLastError

```
void TWAIN_ReportLastError(string pzMsg)
```

Like TWAIN_ErrorBox, but if some details are available from TWAIN about the last failure, they are included in the message box. This function uses TWAIN_LastErrorText to find out about the last error – see below.

TWAIN_LastErrorCode

```
int TWAIN_LastErrorCode(void)
```

Return the most recent EZTwain error code, one of the EZTEC_ codes – See the EZTwain declaration file for your programming language, or refer to eztwain.h.

TWAIN_LastErrorText / TWAIN_GetLastErrorText

```
void TWAIN_GetLastErrorText(LPSTR pzMsg)
string TWAIN_LastErrorText(void)
```

Returns a text string describing the last error encountered by EZTwain. In other words, this function is like TWAIN_LastErrorCode, but it translates the error into a human-readable (English) string. For example, if you try to scan from a device that is disconnected, this function may return something like: "Could not open TWAIN device: EPSON TWAIN 5\n(check power and connections.)". This string may contain end-of-line characters. The returned string will not exceed 512 (ASCII) characters long – if you use TWAIN_GetLastErrorText, make sure you pre-allocate the variable to have enough room.

TWAIN_RecordError

```
void TWAIN_RecordError(int code, string note)
```

Set the internal EZTwain error code, if it is not set already.
This sets the error information that is reported by LastErrorCode, LastErrorText, ReportLastError, and so on. Normally EZTwain records errors internally, but in special circumstances an application might need record an error 'as if' it was an EZTwain internal error.

The error code can be cleared by TWAIN_ClearError, and a few other functions also clear it.

TWAIN_ClearError

```
void TWAIN_ClearError(void)
```

Set the EZTwain internal error code to EZTEC_NONE and clears the last error text.

TWAIN_GetResultCode

```
unsigned TWAIN_GetResultCode(void)
```

Return the result code (TWRC_xxx) from the last triplet sent to TWAIN

TWAIN_GetConditionCode

```
unsigned TWAIN_GetConditionCode(void)
```

Return the condition code from the last triplet sent to TWAIN. (To be precise, from the last call to TWAIN_DS) If no Source is open, return the condition code of the source manager.

TWAIN_ErrorBox

```
void TWAIN_ErrorBox(string pzMsg)
```

Post an error message box with an OK button. The string argument is used as the text of the box, and the application title (see TWAIN_RegisterApp and TWAIN_SetAppTitle) is used as the title or caption of the box. If messages are suppressed (see below) this function does nothing.

Logging

TWAIN_LogFile

```
void TWAIN_LogFile(int fLog)
```

EZTwain can write a quite detailed log of its activity, including every TWAIN call it makes and the result. Log output goes by default to **c:\eztwain.log**. if that directory is writable, otherwise to %TEMP%\eztwain.log
Functions below can change the name and/or directory for logging.

TWAIN_LogFile(0) close log file and turn off logging
TWAIN_LogFile(1) open log file (if not already) and start logging.

If logging is already turned on, TWAIN_LogFile(1) flushes the logfile to disk so prior output won't be lost in a subsequent crash.

TWAIN_WriteToLog

```
void TWAIN_WriteToLog(string pzText)
```

Write text to the EZTwain log file. If the text does not end with an end-of-line character, one is added. If logging is turned off, this call has no effect.

TWAIN_SetLogName

```
BOOL TWAIN_SetLogName(string pzName)
```

Set the filename or path & filename of the EZTwain log file.
If there is a log file open, it is closed, renamed and re-opened.
The default extension is ".log", the default log filename is "eztwain.log".

You can specify a fully-qualified filename, which changes both the folder and filename for logging:

```
TWAIN_SetLogName("c:\temp\scan2tape.log")
```

TWAIN_LogFileName

```
string TWAIN_LogFileName(void)
```

Return the (fully qualified) file path and name for logging.

TWAIN_SetLogFolder

```
void TWAIN_SetLogFolder(string pzFolder)
```

Set the directory that will contain the log file.

Calling this with the empty string resets the log folder to the default (c:\ if it is writable, otherwise %TEMP%) TWAIN_SetLogFolder("c:\ top\middle\logs") will create the 'logs' folder if necessary, but will not create the 'top' or 'middle' folders.
If there is a log file open, it is closed, moved and re-opened.

Functions – TWAIN State

To work with TWAIN, you must have some understanding of the TWAIN *State*. In TWAIN, a conversation with a device moves up and down a ladder of 7 distinct states. In each state, certain operations are permitted and others are invalid, and certain operations or events indicate that the conversation has moved into another state.

TWAIN States

State	EZTwain Symbolic Name	Description
1	TWAIN_PRESESSION	Source Manager not loaded
2	TWAIN_SM_LOADED	Source Manager loaded
3	TWAIN_SM_OPEN	Source Manager open
4	TWAIN_SOURCE_OPEN	Source open for negotiation
5	TWAIN_SOURCE_ENABLED	Source enabled to acquire
6	TWAIN_TRANSFER_READY	Image ready to transfer
7	TWAIN_TRANSFERRING	Image in transit

EZTwain carefully tracks the TWAIN State, and hides a lot of the details of managing the state, but not all. The following group of functions are the ones concerned with directly reading and modifying the TWAIN State.

TWAIN_State

```
int TWAIN_State(void)
```

Returns the State (see above) of the TWAIN conversation.

TWAIN_IsDone

```
BOOL TWAIN_IsDone()
```

Returns FALSE(0) if there is a device open and it is in a state where more scans are available or could be requested. Otherwise returns TRUE (1).

Informally, TRUE means 'stop asking for images' and FALSE means something like 'It would be appropriate at this time to request another image.'
I know, it sounds bizarre, but that's actually how TWAIN works.

This function is designed to be the test at the *bottom* of a do-until loop:

```
If TWAIN_OpenDefaultSource() Then
   TWAIN_SetMultiTransfer(1)
   Do
      TWAIN_AcquireToFilename(0, NextFileName())
   Until TWAIN_IsDone()
   TWAIN_CloseSource()
End If
```

TWAIN_LoadSourceManager

```
int TWAIN_LoadSourceManager(void)
```

Finds and loads the TWAIN Source Manager.
If Source Manager is already loaded, does nothing and returns TRUE(1).
This can fail if TWAIN Source Manager is not installed (in the right place), or if the library cannot load for some reason (insufficient memory?) or if it has been corrupted.

For example, in Windows Server 2008 R2, the TWAIN Source Manager is not installed by default, it is part of something called the Desktop Experience Pack.

TWAIN_OpenSourceManager

```
int TWAIN_OpenSourceManager(HWND hwnd)
```

Opens the Source Manager, if not already open.
If the Source Manager is already open, does nothing and returns TRUE.
This call will fail if the Source Manager is not loaded.

TWAIN_OpenDefaultSource

```
int TWAIN_OpenDefaultSource(void)
```

This opens the source selected in the Select Source dialog.
If some source is already open, does nothing and returns TRUE.
Will load and open the Source Manager if needed.
If this call returns TRUE, TWAIN is in State 4 (TWAIN_SOURCE_OPEN)

TWAIN_OpenSource

```
int TWAIN_OpenSource(string pzName)
```

Opens the Source with the given name.
If that source is already open, does nothing and returns TRUE. If another source is open, closes it and attempts to open the specified source. Will load and open the Source Manager if needed.
If this call returns TRUE, TWAIN is in State 4 (TWAIN_SOURCE_OPEN)

TWAIN_EnableSource

```
int TWAIN_EnableSource(HWND hwnd)
```

Enables the open Source for image acquisition. 'Enabled' in TWAIN parlance means that the Source has permission to begin acquiring images. Until it is enabled, a Source will never begin any image acquisition, nor will it offer an image for transfer.

This call returns TRUE(1) if it leaves the Source in State 5 *or higher.* A return of FALSE(0) indicates that either the Enable failed, or that the Source asked to be closed immediately! If the Source is already enabled when you make this call, it does nothing and returns TRUE.

By default the Source is asked to display its user interface, but this can be controlled with TWAIN_SetHideUI. If a Source is enabled without its user interface, it should if possible immediately offer to transfer an image – on return from TWAIN_EnableSource, the TWAIN_State() should be 6 (Transfer Ready.)

By default the parent window is not affected, but this can be changed using TWAIN_DisableParent.

TWAIN_DisableSource

```
int TWAIN_DisableSource(void)
```

Disables the open Source, if any.
This closes the Source's user interface.
If there is not an enabled Source, does nothing and returns TRUE.

TWAIN_CloseSource

```
int TWAIN_CloseSource(void)
```

Closes the open Source, if any.
If the Source is enabled, disables it first.
If there is not an open Source, does nothing and returns TRUE.

TWAIN_UnloadSourceManager

```
int TWAIN_UnloadSourceManager(void)
```

Closes and unloads the TWAIN Source Manager. If necessary, it will abort transfers, close the open Source if any, and close the Source Manager.
If successful, it returns 1 (TRUE) otherwise 0 (FALSE).

TWAIN_EndXfer

```
int TWAIN_EndXfer(void)
```

Only valid in State 7, it signals the DS to go to either State 6 if it has more transfers ready, or to State 5 if it does not.

It would be very unusual to need to call this: The Acquire functions call this after each transfer. The other state-changing functions will call this if they find themselves in State 7 and need to move down.

TWAIN_AbortAllPendingXfers

```
int TWAIN_AbortAllPendingXfers(void)
```

Functions – Capability

TWAIN_SetXferCount

```
int TWAIN_SetXferCount(int nXfers)
```

Tell the Source the number of images the application will accept.
nXfers = -1 means any number (the default, when a device is opened.)
Returns: 1 for success, 0 for failure.

TWAIN_GetCurrentUnits

```
int TWAIN_GetCurrentUnits(void)
```

Return the current unit of measure: inches, cm, pixels, etc. – see list below. Many TWAIN parameters such as resolution are set and returned in the current unit of measure. There is no error return - in case of error it returns 0 (TWUN_INCHES)

TWAIN unit codes (from twain.h)

#define TWUN_INCHES	0
#define TWUN_CENTIMETERS	1
#define TWUN_PICAS	2
#define TWUN_POINTS	3
#define TWUN_TWIPS	4
#define TWUN_PIXELS	5

TWAIN_SetUnits

```
int TWAIN_SetUnits(int nUnits)
```

Set the current unit of measure for the source. Common unit codes are given above.

- Most sources do not support all units, some support *only* inches. Some cameras support only pixels.
- If you want to get or set resolution in DPI, make sure the current units are inches, or you might get Dots-Per-Centimeter!

TWAIN_GetPixelType

```
int TWAIN_GetPixelType(void)
```

Ask the open device for the current pixel type. See table below.
If anything goes wrong (it shouldn't), this function returns 0 (TWPT_BW).

TWAIN_SetPixelType

```
int TWAIN_SetPixelType(int nPixType)
```

Try to set the current pixel type for acquisition.
The source may select this pixel type, but don't assume it will.
This function should be used in place of the older TWAIN_SetCurrentPixelType.

Pixel Type Codes (TWPT_*)

Code	TWAIN Name	Description
0	TWPT_BW	1-bit per pixel, black and white
1	TWPT_GRAY	grayscale, 8 or 4-bit
2	TWPT_RGB	RGB color, 24-bit (rarely, 15,16,32-bit)
3	TWPT_PALETTE	indexed color (image has a color table) 8 or 4-bit.
4	TWPT_CMY	CMY color, 24-bit
5	TWPT_CMYK	CMYK color, 32-bit

TWAIN_GetBitDepth

```
int TWAIN_GetBitDepth(void)
```

Get the current scanning bit depth, which can depend on the current PixelType.
In theory, bit depth is per color channel e.g. 24-bit RGB has bit depth 8. In practice a *lot* of devices return 24 as the bit depth for RGB. If anything goes wrong, this function returns 0.

TWAIN_SetBitDepth

```
int TWAIN_SetBitDepth(int nBits)
```

Try to set the scanning bit depth for the current pixel type.
Note: You should set a PixelType, and then set the bitdepth for that type.

Tip: Some scanners advertise '12-bit scanning' or '14-bit A/D'. Just because this appears in the scanner specifications does not mean the TWAIN driver supports it, but *if it does*, you usually use it by setting BitDepth to 16. EZTwain should do this automatically, but you may need to also specify Memory Transfer Mode, by calling TWAIN_SetXferMech.

TWAIN_GetCurrent Resolution

```
double TWAIN_GetCurrentResolution(void)
```

Ask the source for the current (horizontal) scanning resolution.
Resolution is in dots per current unit! (See TWAIN_GetCurrentUnits above)
If anything goes wrong (it shouldn't) this function returns 0.0

TWAIN_GetYResolution

```
double TWAIN_GetYResolution(void)
```

Returns the current vertical resolution, in dots per *current unit*.
In the event of failure, returns 0.0.

TWAIN_SetResolution/TWAIN_SetResolutionInt

```
int TWAIN_SetResolution(double dRes)
int TWAIN_SetResolutionInt(int nRes)
```

Try to set the current resolution for scanning. This call sets both vertical (y) and horizontal (x) resolution to the same value. See TWAIN_SetXResolution below if you want to set x and y resolution separately. With scanners if this call succeeds, subsequent scans will be made at this resolution. Naturally, this will not be useful on devices that store images ahead of time, like digital cameras. And generally video-capture devices ignore or reject resolution settings.

Resolution is in dots (samples) per current unit. (See TWAIN_GetCurrentUnits above) The source may select this resolution, but don't assume it will. Almost all scanners that scan paper can support 200 DPI and 300 DPI. Beyond that, the specific values vary quite widely from model to model. To query the values supported by a scanner, see Appendix 2 - Working with Containers, p 161.

TWAIN_SetXResolution / TWAIN_SetYResolution

```
int TWAIN_SetXResolution(double dxRes)
int TWAIN_SetYResolution(double dyRes)
```

Be aware that many scanners will not accept different x and y resolution values - they will either ignore the different y-resolution value, or they will lock the two parameters together: Setting either parameter will set both to the same value.

TWAIN_SetContrast

```
int TWAIN_SetContrast(double dCon)
```

Try to set the current contrast for acquisition. Contrast is *not* a required capability, do not assume a particular scanner supports it.
The TWAIN standard *says* that the range for this cap is -1000 ... +1000.

TWAIN_SetBrightness

```
int TWAIN_SetBrightness(double dBri)
```

Try to set the current brightness for acquisition. Brightness is *not* a required capability, do not assume a particular scanner supports it.
The standard range for this cap is -1000 ... +1000 – we have seen other ranges…

TWAIN_SetThreshold

```
int TWAIN_SetThreshold(double dThresh)
```

Try to set the threshold (TWAIN: ICAP_THRESHOLD) for black and white scanning. Legal values for threshold are 0 to 255. Returns TRUE(1) for success, FALSE(0) for failure.

If a device is not open (TWAIN_State=4) this function will record an error and fail.

How does Threshold work? In B&W scanning mode, you can think of the scanner as measuring each pixel of the document to get an 8-bit number (0..255) where 0 is the darkest measurable black, and 255 is the lightest white. When the scanner delivers the B&W image to your application, pixels that fall below the threshold are set to black, pixels above the threshold are set to white. Nobody can say what happens to pixels that are *equal* to the threshold – don't worry about it!

As you lower the threshold your scanned images tend to become whiter, and as you raise the threshold the scans tend to become darker.

Field Notes

The TWAIN default threshold value is 128, but that does not mean much: Most scanners default to a threshold of 128 when they are opened *but* other scanners seem to default to a stored value, perhaps the last user-selected value.

This setting usually affects only 1-bit scans i.e. PixelType == TWPT_BW.

This setting applies to document scanners, and perhaps film scanners – it is likely to be omitted or ignored by cameras and video digitizers.

A few low-cost scanners will accept this setting without error, but then ignore the value!

As an alternative to setting the scanner threshold, consider scanning in grayscale (pixel type TWPT_GRAY) and converting the images to B&W in software, using EZTwain functions such as DIB_SmartThreshold.

TWAIN_GetCurrentThreshold

```
double TWAIN_GetCurrentThreshold(void)
```

Try to get the current B&W threshold setting – the value of the ICAP_THRESHOLD capability. If this fails for any reason, it will return -1. Note: EZTwain *VERSIONS BEFORE 2.65 RETURNED 128.0*

TWAIN_SetAutoBright

```
int TWAIN_SetAutoBright(BOOL bOn)
```

TWAIN_SetLightPath

```
int TWAIN_SetLightPath(BOOL bTransmissive)
```

Tries to select transparent or reflective media for scanning.
A parameter of TRUE(1) means transparent media (transparency scanning), FALSE(0) means reflective media.
A return of TRUE(1) implies success, FALSE(0) means that the Source refused the request.

TWAIN_SetGamma

```
int TWAIN_SetGamma(double dGamma)
```

TWAIN_SetShadow

```
int TWAIN_SetShadow(double d)       // 0..255
```

TWAIN_SetHighlight

```
int TWAIN_SetHighlight(double d)    // 0..255
```

Document Feeder Control

TWAIN_HasFeeder

```
BOOL TWAIN_HasFeeder(void)
```

Return TRUE(1) if the source indicates it has a document feeder, FALSE(0) otherwise. You will need to have a device open for this query to work.

TWAIN_IsFeederSelected

```
BOOL TWAIN_IsFeederSelected(void)
```

Return TRUE(1) if the document feeder is selected. A device must be open.

TWAIN_SelectFeeder

```
int TWAIN_SelectFeeder(int fYes)
```

(Try to) select or deselect the document feeder.
Return TRUE(1) if successful, FALSE(0) otherwise.

TWAIN_IsAutoFeedOn

```
BOOL TWAIN_IsAutoFeedOn(void)
```

Return TRUE(1) if automatic feeding is enabled, otherwise FALSE(0).
Make sure the feeder is selected before calling this function.

TWAIN_SetAutoFeed

```
int TWAIN_SetAutoFeed(int fYes)
```

(Try to) turn on/off automatic feeding thru the feeder.
Return TRUE(1) if successful, FALSE(0) otherwise.

TWAIN_SetAutoScan

```
int TWAIN_SetAutoScan(int fYes)
```

(Try to) turn on/off scan-ahead (CAP_AUTOSCAN).
Returns TRUE(1) if successful, FALSE(0) otherwise.
This is an optional feature supported by some ADF scanners. When enabled, the scanner will scan pages before they are requested, buffering them in the scanner or host PC. When disabled, the scanner will not feed and scan a page until the application asks for it. Used to achieve maximum throughput on ADF scanners.

TWAIN_IsFeederLoaded

```
BOOL TWAIN_IsFeederLoaded(void)
```

Return TRUE(1) if there are documents in the feeder.
Make sure the feeder is selected before calling this function.

Controlling Duplex Mode

Many document scanners are capable of *duplex* scanning – scanning both sides of an original. Scanning just one side of each page is called *simplex* scanning.

The way duplex scanning works surprises some people, although it is elegant and logical: Duplex scanning treats each page as if it were two pages being scanned single-sided. If you scan 5 pages in duplex mode, the data transfer and the TWAIN activity is essentially the same as scanning 10 pages in simplex mode.

This means that *all duplex scanning is multipage scanning*. If you allow duplex scanning we recommend that you use one of the Multi-image Scanning Functions: TWAIN_AcquireMultipageFile, TWAIN_AcquireArray, TWAIN_AcquirePagesToFiles, etc.

TWAIN_GetDuplexSupport

```
int TWAIN_GetDuplexSupport(void)
```

Query the device for duplex scanning support.
Return values:

0 = no support (or error, or query not recognized)
1 = 1-pass duplex
2 = 2-pass duplex

TWAIN_EnableDuplex

```
int TWAIN_EnableDuplex(int fYes)
```

Enable (fYes=1) or disable (fYes=0) duplex scanning.
Returns TRUE(1) if successful, FALSE(0) otherwise.

TWAIN_IsDuplexEnabled

```
BOOL TWAIN_IsDuplexEnabled(void)
```

Returns TRUE(1) if the device supports duplex scanning and duplex scanning is enabled. FALSE(0) otherwise.

Other Settings

TWAIN_HasControllableUI

```
int TWAIN_HasControllableUI(void)
```

Return 1 if source claims UI can be hidden (see SetHideUI above)
Return 0 if source says UI *cannot* be hidden
Return -1 if source (pre TWAIN 1.6) cannot answer the question.

TWAIN_SetIndicators

```
int TWAIN_SetIndicators(BOOL bVisible)
```

Tell the source to show (hide) progress indicators during acquisition.

TWAIN_SetXferMech / TWAIN_XferMech

```
int TWAIN_SetXferMech(int mech)
int TWAIN_XferMech(void)
```

Try to set or get the transfer mode - one of the following:

```
#define XFERMECH_NATIVE          0
#define XFERMECH_FILE      1
#define XFERMECH_MEMORY    2
```

Normally you do not need to set this mode – the Acquire functions will select the transfer mode based on the scan settings and scanner model.

Note: In the unusual case that you want to transfer 16-bit per channel images (16-bit grayscale or 48-bit color), and assuming your scanner supports it, you should specify *memory transfer mode*. Any resulting 'deep' images in DIB format will not be understood by most software (including Windows) – but most of EZTwain's DIB_ functions will operate on these deep DIBs, and they can be written to TIFF, which is about the only image format that can hold them.

TWAIN_SupportsFileXfer

```
int TWAIN_SupportsFileXfer(void)
```

Returns TRUE(1) if the open Source claims to support file transfer mode (XFERMECH_FILE)
This mode is optional. If TRUE, you can use AcquireFile.

TWAIN_SetCompression / TWAIN_Compression

```
int TWAIN_Compression(void)
int TWAIN_SetCompression(int compression)
```

Set/Return compression format for image transfer from the source device. See twain.h for TWCP_xxx values to use with this capability.

The meaning of this capability depends (in theory) on the current transfer mode (see TWAIN_XferMech above).

If the transfer mode is *File*, this capability should correspond to the compression that will be used in the transferred file.

If the transfer mode is *Memory*, then this is how the incoming buffers of data will be compressed.

For *Native* mode transfers, which is the default for most EZTwain Acquire functions, compression is not supported and will be disabled automatically.

However: TWAIN_AcquireMemory defaults to using Memory transfer mode, and you can also specify Memory transfer mode for the general Acquire functions, by calling e.g.

TWAIN_SetXferMech(XFERMECH_MEMORY)

In Memory transfer mode, you can specify a supported compression and EZTwain will try to transfer images from the scanner using that compression, and will retain the image data in that compression format even in DIB form. When compressed images are written to a file format that uses the same compression, EZTwain will (mostly) write the image to file without decompressing & recompressing it.

All of which means that potentially, if the scanner can deliver JPEG-compressed images and you are writing scanned images to JPEG (or PDF) files, you can arrange to move images from scanner to memory to disk in compressed form, with much lower demand on CPU, RAM, scanner bandwidth and disk bandwidth.

This becomes interesting for grayscale and color images when working with a scanner that can deliver, say, more than 20-30 images/minute, although the threshold depends very much on scanner connection, scanning resolution, computer speed, and disk bandwidth.

B&W images are so much 'lighter' to transfer and process that this optimization is normally not worth the trouble for them.

Raw Capability Get & Set

Note that ordinarily capabilities can be Set or Reset only in State 4 (Source Open) and can be read (Get, GetCurrent, GetDefault) only in State 4 or higher.

TWAIN_Get

```
HCONTAINER TWAIN_Get(unsigned uCap)
```

Issue a DAT_CAPABILITY/MSG_GET to the open source.
Return a capability 'container' - the 'MSG_GET' value of the capability.
Use CONTAINER_* functions to examine and modify the container object.
Use CONTAINER_Free to release it when you are done with it.
A return value of 0 indicates failure: Call GetConditionCode or ReportLastError.

TWAIN_GetDefault

```
HCONTAINER TWAIN_GetDefault(unsigned uCap)
```

Issue a DAT_CAPABILITY/MSG_GETDEFAULT, to get the default value of the specified capability.

TWAIN_GetCurrent

```
HCONTAINER TWAIN_GetCurrent(unsigned uCap)
```

Issue a DAT_CAPABILITY/MSG_GETCURRENT to get the current value of the specified capability. **Caution:** A few Sources will not respond to GetCurrent, but only Get – implying that Get represents the current value.

TWAIN_Set

```
int TWAIN_Set(unsigned uCap, HCONTAINER hcon)
```

Issue a DAT_CAPABILITY/MSG_SET to the open source, using the specified capability and container. Returns 1 (TRUE) if successful, 0 (FALSE) otherwise.

TWAIN_Reset

```
int TWAIN_Reset(unsigned uCap)
```

Issue a MSG_RESET, which should reset the specified capability to its default value.
Returns 1 (TRUE) if successful, 0 (FALSE) otherwise.

TWAIN_GetCapBool

```
BOOL TWAIN_GetCapBool(unsigned cap, BOOL bDefault)
```

Issue a DAT_CAPABILITY/MSG_GETCURRENT on the specified capability to get the value as a BOOL.
Returns the capability value if successful, otherwise returns bDefault.

TWAIN_GetCapFix32

```
double TWAIN_GetCapFix32(unsigned cap, double dDefault)
```

TWAIN_GetCapUint16

```
int TWAIN_GetCapUint16(unsigned cap, int nDefault)
```

TWAIN_SetCapability

```
int TWAIN_SetCapability(unsigned Cap, double dVal)
```

Set the value of a capability of unknown type - such as a custom (proprietary) capability. This is like TWAIN_SetCapOneValue, but you don't have to look up or discover the ItemType. Only useful on capabilities that have a simple current value that is an integer or fractional number. Only valid in State 4.
Return Value: TRUE (1) if successful, FALSE (0) otherwise.
Example:

```
' Tell Canon DR2080 to skip blank pages:
TWAIN_SetCapability(&H8001, 1)
```

TWAIN_SetCapBool

```
int TWAIN_SetCapBool(unsigned Cap, BOOL bVal)
```

Set the value of a capability that has type TWTY_BOOL. Only valid in State 4.
Return Value: TRUE (1) if successful, FALSE (0) otherwise.

TWAIN_SetCapOneValue

```
int TWAIN_SetCapOneValue(unsigned Cap, unsigned ItemType, long
ItemVal)
```

Do a DAT_CAPABILITY/MSG_SET, on capability 'Cap' (e.g. ICAP_PIXELTYPE, CAP_AUTOFEED, etc.) using a TW_ONEVALUE container with the given item type and value. Use SetCapFix32 for capabilities that take a FIX32 value, use SetCapOneValue for the various ints and uints. These functions do not support FRAME or STR items.
Return Value: TRUE (1) if successful, FALSE (0) otherwise.

TWAIN_SetCapFix32 / TWAIN_SetCapFix32R

```
int TWAIN_SetCapFix32(unsigned Cap, double dVal)
int TWAIN_SetCapFix32R(unsigned Cap, int Num, int Den)
```

Do a DAT_CAPABILITY/MSG_SET on capability *Cap* using a TW_ONEVALUE container of a FIX32 item.
Return Value: TRUE (1) if successful, FALSE (0) otherwise.

Use SetCapFix32 and SetCapFix32R for capabilities that take a FIX32 value.

SetCapFix32R uses the value dVal = Num/Den.
This is useful for languages that make it hard to pass double parameters.

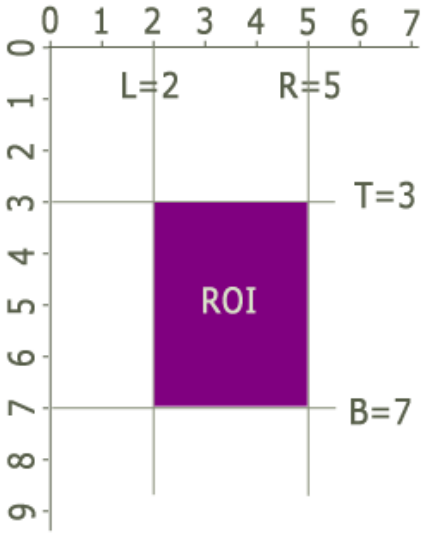
Region of Interest (ROI)

In the jargon of imaging, region-of-interest (ROI) means a particular rectangular region to be processed, out of a larger image area. If you have used a flatbed scanner, you have seen this concept in the scanner's user interface: Within the full scanning area, you can select a smaller rectangle to scan.

TWAIN_SetRegion

```
void TWAIN_SetRegion(double L, double T, double R, double B)
```

This is the most general and powerful function in EZTwain for selecting a region-of-interest. It will try to use the region-scanning abilities of the device, but if the device can't or won't scan the specified region, EZTwain crops each incoming image to the specified area.



Caution: The parameters are NOT x, y, width and height – they are left, top, right, and bottom of the area to scan, measured in the current unit of measure from the top-left corner of the 'original page'. See the diagram to the right.

Caution: Some devices (For example, some Fujitsu fi-series) remember the last paper size selected in their user interface, and will not accept a region setting outside that paper size. To avoid this problem, call TWAIN_SetPaperSize before calling TWAIN_SetRegion.

Example: The following code, with an 8.5" x 11" flatbed scanner, will scan a 3" x 4" square towards the center of the platen, in color at 300dpi, and save it as a TIFF file:

```
If (TWAIN_OpenDefaultSource()) {
    TWAIN_SetUnits(TWUN_INCHES);
    TWAIN_SetResolution(300);
    TWAIN_SetPixelType(TWPT_RGB);
    TWAIN_SetRegion(2.0, 3.0, 5.0, 7.0);
    // scan starts 2" from left side, 3" from top
    // scan stops 5" from left side, 7" from top.
    // scan is 5.0-2.0 = 3" wide and 7.0-3.0 = 4" high
    TWAIN_SetHideUI(1);
    TWAIN_AcquireToFilename(0, "myfile.tif");
}
```

TWAIN_ResetRegion

```
void TWAIN_ResetRegion(void)
```

Resets the region set with TWAIN_SetRegion, so that EZTwain stops trying to set or crop to a region of interest.

TWAIN_SetImageLayout

```
int TWAIN_SetImageLayout(double L, double T, double R, double B)
```

Lower-level region-of-interest (ROI) function. Set the area to scan, using DAT_IMAGELAYOUT/MSG_SET.
Note: Even though the TWAIN standard lists this feature as required most cameras ignore it, along with some ADF scanners and other devices. This call is only valid in State 4 that is, when a device is open.

L, T, R, B = distance to left, top, right, and bottom edge respectively of area to scan, measured in the current unit of measure from the top-left corner of the 'original page' (TWAIN 1.6 8-22). See the warning below about units.

Returns TRUE (1) if successful, FALSE (0) otherwise. Common causes of failure:

1. Not in State 4 / Source open. See TWAIN_OpenDefaultSource.
2. The device does not support image layout.
3. Incorrect parameters – see the example below.

Do not assume that image layout is pixel-precise. Many devices deliver images that differ from the requested image layout by a few pixels in width or height.

In theory the numbers used in image layout are measurements in the current unit of measure (see TWAIN_GetCurrentUnits / TWAIN_SetUnits.)

In practice quite a few TWAIN devices ignore the unit setting and always measure image layout in inches.

TWAIN_GetImageLayout / TWAIN_GetDefaultImageLayout

```
int TWAIN_GetImageLayout(double *L, double *T, double *R, double
*B)
int TWAIN_GetDefaultImageLayout(double *L, double *T, double *R,
double *B)
```

Get the current or default (power-on) area to scan.
See the warning above about units.
This call is valid in States 4-6.
Return value: 1 = success, 0 = failure.

TWAIN_ResetImageLayout

```
int TWAIN_ResetImageLayout(void)
```

Reset the scan area to the default (power-on) settings.
This call is only valid in State 4.
Return value: 1 = success, 0 = failure.

TWAIN_SetFrame

```
int TWAIN_SetFrame(double L, double T, double R, double B)
```

This is an alternative way to set the scan area.
Some scanners will respond to this instead of SetImageLayout.
Return value: 1 = success, 0 = failure.
This call is only valid in State 4, when a Source is open.
L, T, R, B = distance to left, top, right, and bottom edge of the area to scan, measured in the current unit of measure.

TWAIN_SetPaperSize

```
int TWAIN_SetPaperSize(int nPaper)
```

Asks the scanner to scan a specific standard paper size.

Note: Some devices support this, some don't. If the Source refuses to set the requested paper size, TWAIN_SetPaperSize will try the two other ways to select the scan area: TWAIN_SetImageLayout and TWAIN_SetFrame.

Caution: TWAIN defines *no default paper size*. This means that when you open a device, it is free to select whatever paper size it feels like. If you are running a device with its user interface suppressed, we recommend that you call TWAIN_SetPaperSize. This may fail, but when it does the device usually has a reasonable default – such as 8.5 x 11 inches, or the device's maximum scan area.

Standard TWAIN Paper Sizes

Constant Name	Value
PAPER_NONE	0
PAPER_A4LETTER	1
PAPER_B5LETTER	2
PAPER_USLETTER	3
PAPER_USLEGAL	4
PAPER_A5	5
PAPER_B4	6
PAPER_B6	7
PAPER_USLEDGER	9
PAPER_USEXECUTIVE	10
PAPER_A3	11
PAPER_B3	12
PAPER_A6	13
PAPER_C4	14
PAPER_C5	15
PAPER_C6	16
PAPER_4A0	17
PAPER_2A0	18
PAPER_A0	19
PAPER_A1	20
PAPER_A2	21

PAPER_A7	22
PAPER_A8	23
PAPER_A9	24
PAPER_A10	25
PAPER_ISOB0	26
PAPER_ISOB1	27
PAPER_ISOB2	28
PAPER_ISOB5	29
PAPER_ISOB7	30
PAPER_ISOB8	31
PAPER_ISOB9	32
PAPER_ISOB10	33
PAPER_JISB0	34
PAPER_JISB1	35
PAPER_JISB2	36
PAPER_JISB3	37
PAPER_JISB4	38
PAPER_JISB6	39
PAPER_JISB7	40
PAPER_JISB8	41
PAPER_JISB9	42
PAPER_JISB10	43
PAPER_C0	44
PAPER_C1	45
PAPER_C2	46
PAPER_C3	47
PAPER_C7	48
PAPER_C8	49
PAPER_C9	50
PAPER_C10	51
PAPER_USSTATEMENT	52
PAPER_BUSINESSCARD	53

TWAIN_GetPaperDimensions

```
BOOL TWAIN_GetPaperDimensions(int nPaper, int nUnits,
                              double *pdW, double *pdH)
```

Retrieves the width and height of a standard paper size, in specified units. For nPaper, use one of the PAPER_ codes listed above. For nUnits, use one of the TWUN_ unit codes, such as TWUN_INCHES(0) or TWUN_CENTIMETERS(1). For pdW and pdH, pass *pointers* to 64-bit double-precision floating point variables – or in languages that support passing parameters *by reference*, pass the names of two double-precision float variables.

Returns TRUE(1) if successful. Returns FALSE(0) if nPaper or nUnits are invalid (unrecognized) values.

Tone Control

TWAIN_SetGrayResponse

```
int TWAIN_SetGrayResponse(const long pcurve[256])
```

Define a translation of gray pixel values.
When device digitizes a pixel with value V, that pixel is translated to value pcurve[V] before it is returned to the application.
Caution: Supported by few devices.

- Device must be open (State 4),
- Current PixelType must be TWPT_GRAY or TWPT_RGB,
- current BitDepth should be 8.
- pcurve must be a table (array, vector) of 256 entries.

TWAIN_SetColorResponse

```
int TWAIN_SetColorResponse(const long pred[256], const long
pgreen[256], const long pblue[256])
```

Define a translation of color values.
Like TWAIN_SetGrayResponse (above) but separate translation can be applied to each color channel. Supported by few devices.

TWAIN_ResetGrayResponse/ TWAIN_ResetColorResponse

```
int TWAIN_ResetGrayResponse(void)
int TWAIN_ResetColorResponse(void)
```

These two functions reset the response curve to map every value V to itself i.e. a 'do nothing' translation.

Functions – Settings Dialog

TWAIN_DoSettingsDialog

```
int TWAIN_DoSettingsDialog(HWND hwnd)
```

Display the device's settings dialog and allow the user to adjust any or all settings. This function returns when the user closes the dialog. This feature is *optional* in TWAIN. If a device has a settings dialog, it is normally very similar to the device's scanning dialog, with the [Scan] button replaced with an [OK] button.

To a first approximation, this feature is supported by no cameras, few flatbeds, and not all document scanners.

A device *may* remember its settings when it is closed and re-opened, but it may not: TWAIN does not require this. You can use TWAIN_GetCustomDataToFile (below) to save all the settings of a device, and TWAIN_SetCustomDataFromFile to restore them later. Any device that supports a settings dialog supports the CustomData functions.

If a device is open, uses that device. If no device is currently open, uses the default device.

To check if a device supports this, open the device and call
TWAIN_GetCapBool(CAP_ENABLEDSUIONLY, FALSE)
which should return TRUE(1) if the device supports this feature.

Return values:

1	dialog was displayed and user clicked OK
0	dialog was displayed and user clicked Cancel
-1	dialog not displayed - some error. Call TWAIN_LastErrorCode, ReportLastError, or similar function for more details.

TWAIN_EnableSourceUiOnly

```
int TWAIN_EnableSourceUiOnly(HWND hwnd)
```

This is the underlying 'asynchronous' function for TWAIN_DoSettingsDialog.
Opens the device's settings dialog, if this is supported.
Returns TRUE (1) if successful, FALSE (0) otherwise.

Note: If successful, this call leaves the dialog open, so your program must run a message pump at least until the user closes it. If you don't understand what that means, *don't call this function!* Use TWAIN_DoSettingsDialog, above.

Functions – Custom DS Data

These functions support an optional feature of TWAIN that allows the application to read or write all the settings of a TWAIN device in a single operation. This a good news/bad news feature:

Good: 1. (If implemented properly) it reads and writes *all* settings of the device, even settings that are not accessible through TWAIN, like "melt film after scanning" and "pixel thickness." Or... the imprinter step size and direction on Fujitsu 'fi' series scanners.

2. When supported, it's a great way to set a scanner to a predefined, completely known state.

Bad: 1. It is not universally supported: no cameras that we have seen, not so many flatbeds, not all document scanners.

2. The format of the saved settings is *vendor-specific*. If you parse the data, you are tied to that vendor, and possibly to that specific model and/or device driver.

Support for this feature can be tested when a device is open, by reading the value of the Boolean capability CAP_CUSTOMDSDATA. In C:

```
if (TWAIN_GetCapBool(CAP_CUSTOMDSDATA, FALSE)) {
```

TWAIN_GetCustomDataToFile

```
int TWAIN_GetCustomDataToFile(string pzFile)
```

Takes a file specification, reads the Custom DS Data from the device and saves it in the file. An existing file will be overwritten. Only valid in State 4 – a device must be open. See TWAIN_OpenSource, 120. The device must support this optional feature, see above.
Returns TRUE(1) if successful, FALSE(0) otherwise.

TWAIN_SetCustomDataFromFile

```
int TWAIN_SetCustomDataFromFile(string pzFile)
```

Takes a file specification, reads the Custom DS Data from the file and writes it to the device. The file must exist. Only valid in State 4 – a device must be open. See TWAIN_OpenSource, 120. The device must support this optional feature, see above.
Returns TRUE(1) if successful, FALSE(0) otherwise.

Functions – Container

For theory and practice of using containers, see the section How To: Work with Containers (p 161).

CONTAINER_Free

void CONTAINER_Free(HCONTAINER hcon)
Free the memory and resources of a capability container.

CONTAINER_Copy

```
HCONTAINER CONTAINER_Copy(HCONTAINER hcon)
```
Create an exact copy of the container.

CONTAINER_Equal

```
BOOL CONTAINER_Equal(HCONTAINER hcon1, HCONTAINER hcon2)
```
Return TRUE (1) if all properties of hcon1 and hcon2 are the same. Otherwise return FALSE (0).

CONTAINER_IsValid

```
BOOL CONTAINER_IsValid(HCONTAINER hcon)
```
Returns 1 (TRUE) if the container seems to be valid, 0 (FALSE) if not.
A valid container is one that will not cause errors or exceptions if accessed with the other CONTAINER_ functions.

CONTAINER_Format

```
int CONTAINER_Format(HCONTAINER hcon)
```
Return the 'format' of this container: CONTAINER_ONEVALUE, etc.

Container formats, same codes as in TWAIN.H

CONTAINER_ARRAY	3
CONTAINER_ENUMERATION	4
CONTAINER_ONEVALUE	5
CONTAINER_RANGE	6

CONTAINER_ItemCount

```
int CONTAINER_ItemCount(HCONTAINER hcon)
```
Return the number of values in the container. For a ONEVALUE, return 1.

CONTAINER_ItemType

```
int CONTAINER_ItemType(HCONTAINER hcon)
```
Return the item type (what exact kind of values are in the container.)
See the TWTY_* definitions in TWAIN.H

CONTAINER_TypeSize

```
int CONTAINER_TypeSize(int nItemType)
```
Return the size in bytes of an item of the specified type (TWTY_*)

CONTAINER_FloatValue / CONTAINER_IntValue

```
double CONTAINER_FloatValue(HCONTAINER hcon, int n)
int CONTAINER_IntValue(HCONTAINER hcon, int n)
```

Return the value of the nth item in the container.
n is forced into the range 0 to ItemCount(hcon)-1.

CONTAINER_StringValue / CONTAINER_GetStringValue

```
string CONTAINER_StringValue(HCONTAINER hcon, int n)
void CONTAINER_GetStringValue(HCONTAINER hcon, int n, LPSTR
pzText)
```

Return the nth value in a container, in the form of a string.
The first form is a function that returns the string as its value. (Not available in VB.)
The second form expects a string variable as its 3rd parameter - in most languages, the string must be pre-allocated with enough space to hold the returned value - see
n is forced into the range 0 to ItemCount(hcon)-1.

CONTAINER_ValuePtr

```
BYTE* CONTAINER_ValuePtr(HCONTAINER hcon, int n)
```

CONTAINER_ContainsValue

```
int CONTAINER_ContainsValue(HCONTAINER hcon, double d)
```

Return 1 (TRUE) if the value d is one of the items in the container.

CONTAINER_FindValue

```
int CONTAINER_FindValue(HCONTAINER hcon, double d)
```

Return the index of the value d in the container, or -1 if not found.

CONTAINER_CurrentValue / CONTAINER_DefaultValue

```
double CONTAINER_CurrentValue(HCONTAINER hcon)
double CONTAINER_DefaultValue(HCONTAINER hcon)
```

Return the container's current or power-up (default) value.
Array containers do not have these and will return -1.0.
OneValue containers always return their (one) value.

CONTAINER_CurrentIndex / CONTAINER_DefaultIndex

```
int CONTAINER_DefaultIndex(HCONTAINER hcon)
int CONTAINER_CurrentIndex(HCONTAINER hcon)
```

Return the index of the Default or Current value, in an Enumeration.
Return -1 if the container is not an Enumeration.

CONTAINER_MinValue / CONTAINER_MaxValue

```
double CONTAINER_MinValue(HCONTAINER hcon)
double CONTAINER_MaxValue(HCONTAINER hcon)
```

Return the minimum or maximum value of all the values in a container.
Return -1 if the container contains no values, or the values are not scalars.

CONTAINER_StepSize

```
double CONTAINER_StepSize(HCONTAINER hcon)
```

Return the *step* value of a Range container.
Returns -1.0 if the container is not a Range.

CONTAINER_OneValue / CONTAINER_Array

```
HCONTAINER CONTAINER_OneValue (int nItemType, double dVal)
HCONTAINER CONTAINER_Array (int nItemType, int nItems)
```

These functions create containers from scratch:
nItemType is one of the TWTY_* item types from TWAIN.H
nItems is the number of items, in an array or enumeration.

CONTAINER_Range / CONTAINER_Enumeration

```
HCONTAINER CONTAINER_Range(int nItemType, double dMin, double
dMax, double dStep)
HCONTAINER CONTAINER_Enumeration(int nItemType, int nItems)
```

These functions create containers from scratch:
nItemType is one of the TWTY_* item types from TWAIN.H
nItems is the number of items, in an array or enumeration.
dMin, dMax, dStep are the beginning, ending, and step value of a range.

CONTAINER_SetItem / CONTAINER_SetItemString

```
int CONTAINER_SetItem(HCONTAINER hcon, int n, double dVal)
int CONTAINER_SetItemString(HCONTAINER hcon, int n, LPCTSTR
pzVal)
```

Set the nth item of the container to dVal or pzText.
NOTE: A OneValue is treated as an array with 1 element.
Return 1 (TRUE) if successful. 0 (FALSE) for failure:
  The container is not an array, enumeration, or onevalue
  n < 0 or n >= CONTAINER_ItemCount(hcon)
  the value cannot be represented in this container's ItemType.

CONTAINER_SetItemFrame

```
int CONTAINER_SetItemFrame(HCONTAINER hcon, int n, double l,
double t, double r, double b)
```

Set the nth item of the container to frame(l,t,r,b).
NOTE: A OneValue is treated as an array with 1 element.
Return 1 (TRUE) if successful. 0 (FALSE) for failure:
  The container is not an array, enumeration, or onevalue
  n < 0 or n >= CONTAINER_ItemCount(hcon)
  the value cannot be represented in this container's ItemType.

CONTAINER_SelectDefaultValue / CONTAINER_SelectDefaultItem

```
int CONTAINER_SelectDefaultValue(HCONTAINER hcon, double dVal)
int CONTAINER_SelectDefaultItem(HCONTAINER hcon, int n)
```

CONTAINER_SelectCurrentValue / CONTAINER_SelectCurrentItem

```
int CONTAINER_SelectCurrentValue(HCONTAINER hcon, double dVal)
int CONTAINER_SelectCurrentItem(HCONTAINER hcon, int n)
```

Select the current or default value within an enumeration or range, by specifying either the value, or its index.
Returns 1 (TRUE) if successful, 0 (FALSE) otherwise.
This will fail if:

- The container is not an enumeration or range.
- dVal is not one of the values in the container
- n < 0 or n >= CONTAINER_ItemCount(hcon)

CONTAINER_DeleteItem

```
int CONTAINER_DeleteItem(HCONTAINER hcon, int n)
```

Delete the nth item from an Array or Enumeration container.
Returns 1 (TRUE) for success, 0 (FALSE) otherwise. Failure causes:

- invalid container handle
- container is not an array or enumeration
- n < 0 or n >= ItemCount(hcon)

CONTAINER_InsertItem

```
int CONTAINER_InsertItem(HCONTAINER hcon, int n, double dVal)
```

Insert an item with value dVal into the container at position n.
If n = -1, the item is inserted at the end of the container.

Functions – Testing & Validation

TWAIN_Testing123

```
HANDLE TWAIN_Testing123(string pz, int n, HANDLE h, double d,
unsigned u)
```

Display a dialog box with the parameter values in it. Use this to test that you can call EZTwain and pass parameters correctly. It returns the value of the HANDLE h parameter.

TWAIN_SelfTest

```
int TWAIN_SelfTest(unsigned f)
```

Perform internal self-test.
Parameters

f	ignored for now

Return Values

0	success
other	internal test failed.

Functions – Obscure (Even for TWAIN)

TWAIN_AutoClickButton

```
Void TWAIN_AutoClickButton(string pzButtonName)
```

This odd little function can be used, sometimes, to automate image transfers from a device that *insists* on displaying a user interface dialog. Call this function before calling an Acquire function. When the Acquire starts and the device dialog pops up, EZTwain will search the dialog for a button with the specified name and simulate the user clicking that button. If you pass a null string to this function, it looks for a button with one of the common (English) labels: "Scan", "Capture", "Start Scan", "Take Picture", "Scan Now". Case differences are ignored ('A' is the same as 'a') as are the underlined letters in some button labels.

TWAIN_RegisterApp

```
void TWAIN_RegisterApp (
   int   nMajorNum, // version numbers are treated as
   int   nMinorNum, // nMajorNum.nMinorNum
   int   nLanguage, // language (see TWLG_xxx in TWAIN.H)
   int   nCountry,  // country (see TWCY_xxx in TWAIN.H)
   string     lpszVer,   // version as string e.g. "1.0b3 beta"
   string     lpszMfg,   // vendor e.g. "Zzzzip Software"
   string     lpszFam,   // product family e.g. "Whooshy"
   string     lpszApp)   // specific product e.g. "Whooshy Paint"
```

This is the long form of TWAIN_SetAppTitle, and need only be used if you know that some Source needs the additional information. TWAIN_RegisterApp should be called as one of the first EZTwain calls.

TWAIN_Blocked

```
int TWAIN_Blocked(void)
```

Returns 1 if processing is inside the TWAIN Source Manager or a Source, 0 otherwise. If TWAIN is blocked, EZTwain Pro 2.95 and later will fail any operation that would require a call into TWAIN – otherwise such calls almost always deadlock.

Why do we have this? Because we found that TWAIN drivers sometimes threw uncaught exceptions (divide-by-zero, invalid address) which were not caught by the TWAIN manager, and so ended up being caught by EZTwain. When this happened, the TWAIN manager was left in a kind of 'death trap' state – any call into it would block forever on a serialization semaphore. When this function returns TRUE, it means TWAIN is in that state and is unusable. We use this in applications that have to be robust in the face of bizarre TWAIN failures, such as Twister.

TWAIN_UserClosedSource

```
int TWAIN_UserClosedSource(void)
```

Return TRUE (1) if during the last acquire the user asked the Source to close. 0 otherwise of course. This flag is cleared each time you start any kind of acquire, and it is set if EZTwain receives a MSG_CLOSEDSREQ message through TWAIN.

TWAIN_BuildName

```
char* TWAIN_BuildName(void)
```

Return a string describing the build of EZTwain e.g. "Beta1 Debug"

TWAIN_GetBuildName

```
void TWAIN_GetBuildName(LPSTR psName)
```

TWAIN_AcquireMemoryCallback

BOOL TWAIN_AcquireMemoryCallback(HWND hwnd, MEMXFERCALLBACK cb, LPVOID data)

typedef BOOL (WINAPI *MEMXFERCALLBACK)(LPVOID data);

Like TWAIN_AcquireMemory, but you provide a call-back function. The call-back is called when the transfer is ready, and is responsible for setting up the transfer, transferring the data, and doing clean-up.

TWAIN_SetTiled / TWAIN_Tiled

```
BOOL TWAIN_Tiled(void)
int TWAIN_SetTiled(BOOL bTiled)
```

Set/Return whether source does memory xfer via strips or tiles.
bTiled = TRUE if it uses tiles for transfer.

TWAIN_SetPlanarChunky / TWAIN_PlanarChunky

```
int TWAIN_PlanarChunky(void)
int TWAIN_SetPlanarChunky(int shape)
```

Set/Return current pixel ‘packing’ for memory transfers. See the TWAIN specification for details.

TWAIN_SetPixelFlavor / TWAIN_PixelFlavor

```
int TWAIN_PixelFlavor(void)
int TWAIN_SetPixelFlavor(int flavor)
```

Set/Return pixel 'flavor' - whether 0 is black or white:

#define CHOCOLATE_PIXELS 0 // zero pixel represents darkest shade
#define VANILLA_PIXELS 1 // zero pixel represents lightest shade

TWAIN_GetCapCurrent

```
int TWAIN_GetCapCurrent(unsigned Cap,
                        unsigned ItemType,
                        void FAR *pVal)
```

Do a DAT_CAPABILITY/MSG_GETCURRENT on capability 'Cap'. Copy the current value out of the returned container into *pVal. If the operation fails (the source refuses the request), or if the container is not a ONEVALUE or ENUMERATION, or if the item type of the returned container is incompatible with the expected TWTY_ type in ItemType, returns FALSE. If this function returns FALSE, *pVal is not touched.

TWAIN_ToFix32 / TWAIN_ToFix32R

```
long TWAIN_ToFix32(double d)
```

Convert a floating-point value to a 32-bit TW_FIX32 value that can be passed to e.g. TWAIN_SetCapOneValue.

```
long TWAIN_ToFix32(int Numerator, int Denominator)
```

Convert a rational number to a 32-bit TW_FIX32 value.
Returns a TW_FIX32 value that approximates Numerator/Denominator

TWAIN_Fix32ToFloat

```
double TWAIN_Fix32ToFloat(long nfix)
```

Convert a TW_FIX32 value (as returned from some capability inquiries) to a double (floating point) value.

TWAIN_MessageHook

```
int TWAIN_MessageHook(LPMSG lpmsg)
```

This function detects Windows messages that should be routed to an enabled Source, and picks them off. In a full TWAIN app, TWAIN_MessageHook is called inside the main GetMessage loop, whose skeleton code looks like something like this:

```
MSG msg;
BOOL bGot;
while ((bGot = GetMessage((LPMSG)&msg, NULL, 0, 0)) != 0) {
   if (bGot < 0) {
      // something weird.
   } else if (!TWAIN_MessageHook ((LPMSG)&msg)) {
      TranslateMessage ((LPMSG)&msg);
      DispatchMessage ((LPMSG)&msg);
   }
} // while
```

TWAIN_GetSourceIdentity

```
int TWAIN_GetSourceIdentity(LPVOID ptwid)
```

Get a copy of the TW_IDENTITY structure (see twain.h) used inside EZTwain to hold information about the current / most recently opened Source.

TWAIN_DS

```
int TWAIN_DS(unsigned long DG, unsigned DAT, unsigned MSG, void
FAR *pData)
```

TWAIN_DS passes the triplet (DG, DAT, MSG, pData) to the open Source if any.
Returns 1 (TRUE) if the operation is successful, 0 (FALSE) otherwise.
The last result code can be retrieved with TWAIN_GetResultCode(), and the corresponding condition code can be retrieved with TWAIN_GetConditionCode().
If no source is open this call will fail, result code TWRC_FAILURE, condition code TWCC_NODS.

This function plus TWAIN_Mgr below give you direct access to the TWAIN API, although this function does provide two hidden services: It tracks the TWAIN state, and it traps exceptions inside TWAIN and turns them into failure returns.

TWAIN_Mgr

```
int TWAIN_Mgr(unsigned long DG, unsigned DAT, unsigned MSG, void
FAR *pData)
```

Pass a triplet to the Source Manager (DSM).
Returns 1 (TRUE) for success, 0 (FALSE) otherwise.
See GetResultCode, GetConditionCode, and ReportLastError functions for diagnosing and reporting a TWAIN_Mgr failure.

If the Source Manager is not open, this call fails setting result code TWRC_FAILURE, and condition code=TWCC_SEQERROR (triplet out of sequence).

This function with TWAIN_DS above give you direct access to the TWAIN API.

Functions – Deprecated

For a list of functions that have been deleted from the current release, see Appendix 1 – History.

The following functions are candidates to be removed in a future major-version update to EZTwain Pro:

{no functions at this time}

Glossary

BMP (BitMaP) File

The standard raster image file format used by Microsoft® Windows. BMP files can store 1-bit, 4-bit, 8-bit, and 24-bit (per pixel) images, and much less commonly, 16-bit and 32-bit images. While there is an optional compression for BMP files, it is rarely used and is particularly unsuitable for continuous tone images. A BMP file is simply a DIB (q.v.) with a small header stuck in front.

Capability

TWAIN term for a property or setting of a device, that is accessible ('exposed') to TWAIN applications. Every standard capability is given a name and code.

Some capabilities are read-only, such as CAP_PAPERDETECTABLE, which has the value TRUE or FALSE depending on whether the device can detect paper in its document feeder. A device that has no document feeder will not even have this capability.

Other capabilities can be read and modified by the application, and can have complex behavior: For example, ICAP_PIXELTYPE describes the 'kind of pixels' the device can deliver. Most scanners can deliver 1-bit Black & White, Grayscale, or RGB Color, and will allow the application to specify which it would like, but a webcam may offer only RGB, and reject an attempt to set any other value.

Container

TWAIN term for a global memory block holding one or more values or data items, used to represent a *capability value* i.e. a property or setting of a device.

Datasource (Source, Data Source, DS)

A device-specific TWAIN interface module. In some ways, a Source is just a glorified driver – but a TWAIN Source is different from a conventional driver in three ways:

1. A Source has a *user interface* – typically a dialog box, sometimes an elaborate collection of overlapping windows.
2. A Source runs as part of the client application *i.e. your program.* It is in fact a DLL, and functions as a temporary extension of the application. This is radically different from typical device drivers that run inside the protected shell of the OS.
3. A Source in most cases does not communicate directly with its device, but works through a lower-level kernel-mode driver. Sometimes this driver is device-specific and is provided by the device manufacturer. Sometimes it's a generic Windows driver that simply provides communication services for USB devices, or SCSI devices.

TWAIN Sources install underneath \Windows\twain_32 or \Windows\twain_64 and have an extension of **.ds** even though they are DLLs.

Datasource Manager (AKA DSM)

The common TWAIN module distributed by the TWAIN Working Group. This module, often called the DSM, is maintained by the members of the Working Group, and is provided in binary form to all TWAIN developers. It is customary for TWAIN devices to install/update the DSM when they install their individual TWAIN Sources and low-level drivers.

The job of the DSM is to act as a go-between, coordinating and passing information between TWAIN applications and TWAIN Sources. Many users and developers assume that 'TWAIN' (meaning the DSM) does some kind of serious processing or translation during scanning. This is not true. The DSM's jobs are simple:

1. Find and enumerate the installed Sources, display the Select Source dialog on request, and remember which DS is currently the default.
2. When an application issues an OPEN request, connect it with a Source.
3. Pass TWAIN operation requests from the application to the open DS, and pass notifications back to the application from the DS.

All user interface, image processing, error handling, etc. etc. is divided between the application and the DS.

Default Datasource AKA Default TWAIN Device

To avoid applications asking users 'From which device?' every time they want to acquire an image, TWAIN defines a default device.

If there is only one TWAIN device installed in the system, then - that's it. Otherwise, it is the last TWAIN device selected by the user in the Select Source dialog. Which is the *only* bit of user interface provided by TWAIN itself, so to speak – it is displayed by the TWAIN Source Manager.

Deskew

Scanning jargon meaning 'to straighten up'. If you have done much scanning, you have probably noticed that some documents are scanned at a slight angle, This is called *skew* in the imaging industry. Some scanners, and many image software packages including EZTwain, can straighten out or *deskew* such tilted scans.

DIB (Device Independent Bitmap)

An image format defined and used by Windows – EZTwain stores images in memory as DIBs. A DIB consists of a header giving height, width, bits per pixel, resolution, and so forth, followed by a color table if needed, followed by the pixels of the image. As an added complexity, the convention in Windows is to store DIBs in global memory blocks and work with the *handles* of these blocks – which are *not pointers.* You will frequently see images referenced by objects of type HANDLE, or HGLOBAL or HDIB.
As a consequence, to access the information in a DIB you must either lock the DIB handle to obtain a pointer, or call one of EZTwain's many DIB functions (which lock and unlock the handle internally.)

FIX32 or TW_FIX32

A structure defined by TWAIN to represent fractional numeric values. It is a 'fixed point' representation, a signed 32-bit integer with an implied binary point in the middle - i.e. 16 fractional bits. EZTwain includes various functions for working directly with FIX32 values, but for the most part EZTwain functions accept and return *double* (64-bit floating point) values and convert to FIX32 values internally.

GIF - Graphics Interchange Format

An image format originally defined by CompuServe. The version known as GIF 89a became ubiquitous on the World-Wide Web, although it was impaired by patent issues for many years. Once the patents expired, EZTwain adopted full support for GIF. It is typically used as an export format. GIF has a standard 'zip-like' compression mode, and does well with line-art, text documents, and computer-generated images. It does very poorly at compressing grayscale and color images.

JFIF (JPEG File Interchange Format) File

Technically, the format commonly used to store JPEG-compressed images.

Nobody officially maintains this format, despite its incredibly wide use. The definition is available for example at:
http://www.w3.org/Graphics/JPEG/jfif3.pdf

JPEG (Joint Photographic Experts Group)

A standard form of compression for grayscale and color images. It is a *lossy* form of compression, because it discards some (hopefully less important) information from the original image, based on a model of the Human Visual System. The JPEG committee did not define a specific file format, so JFIF was defined – see JFIF files.

If you are seriously interested in JPEG compression, you can start here:
http://old.jpeg.org/public/jpeghomepage.htm

PDF (Adobe Portable Document Format) File

A widely used format for public distribution of electronic documents, PDF can be used to store single-images or multiple pages. When writing to PDF, EZTwain stores 1-bit images losslessly, but uses lossy JPEG compression for grayscale and color images. See http://www.adobe.com/products/acrobat/main.html

PNG (Portable Network Graphics) File

A relatively new image file format designed to supercede GIF, particularly for World Wide Web graphics: PNG uses a lossless compression that does best on 1-bit B&W line-art (or text), where there are significant areas of identical color. PNG provides little compression for photographs and other imagery. See http://www.libpng.org/pub/png/ for *all* the details.

Resolution

In digital imaging and scanning, *resolution* is how finely a digital image divides up the physical world, commonly measured in dots per inch (DPI). DPI is commonly

used, even in metric countries i.e. outside the USA. You will sometimes encounter the term *samples* as a more highbrow synonym for "dots."

If an image is "200 DPI" this means the image contains 200 rows per vertical inch, and 200 columns per horizontal inch. Compared to a 100 DPI image, the 200 DPI image can distinguish a line or dot that is half as wide. The 200 DPI image also has 4 times as many pixels in it.

Thumbnail

A thumbnail is a small image - typically 32 to 64 pixels high, similar to an icon except that it is a low-resolution copy of the original that it represents.

TIFF (Tagged Image File Format) File

A complex, comprehensive image file format – the dominant standard for high-quality image interchange in graphic arts and publishing.
See: The Unofficial TIFF Home Page at http://home.earthlink.net/~ritter/tiff/

Transfer Mode (AKA Transfer Mechanism AKA Xfer Mech)

TWAIN defines three ways for a TWAIN device (driver) to transfer image data to an application:

1. Native: In a single block in memory, formatted as a DIB (see above).
2. Memory: In a series of buffers, each holding part of the image.
3. File: As a file on disk.

The file transfer mode is optional, and not all TWAIN devices support it. For most scanning applications, you should let EZTwain choose the transfer mode.

Triplet

Specialized TWAIN term: Every communication from an application to TWAIN consists of a triplet of codes plus a pointer. The functions TWAIN_DS and TWAIN_Mgr expose this bottom-level interface to TWAIN. Discussions of the TWAIN protocol often refer to *sending triplets* and *the response to such-and-such triplet*, and the core of the TWAIN Specification is the list of triplet definitions.

TWAIN

The Technology Without An Interesting Name. Actually TWAIN is not an acronym, which has caused endless confusion and frustration over the years.

TWAIN is an industry-standard application programming interface (API) for applications to acquire images from imaging devices. It is only available on Microsoft Windows and Apple Macintosh. Familiar devices such as scanners, digital cameras, and webcams typically support TWAIN, as do some more exotic devices such as x-ray film scanners and digital microscopes. Almost all programs that can work with images support TWAIN – image editors such as Photoshop, web design programs like Dreamweaver. The latest specification (at the time of this writing) is 1.9 – See the TWAIN Specification

TWAIN Working Group

The informal organization that maintains the TWAIN Specification (Version 1.9 at the time of this writing) – The specification and much other TWAIN information is available at www.twain.org

TWAIN Compliance

Compliance with the TWAIN standard is *voluntary*: TWAIN is not a trademark, and the TWAIN Working Group does not certify or enforce compliance of products. So, while it is wonderfully comprehensive and flexible, TWAIN is correspondingly burdened by its complexity, and a great amount of variation between devices. Almost every "TWAIN compliant" device can be shown to be non-compliant in some respect. EZTwain hides as much of this messiness as it can.

TWAIN States

TWAIN defines 7 states of a TWAIN 'conversation'. Certain operations are only valid in certain states, and certain state transitions tell the Source, or the application, when they can *get to work*. Following are definitions of the 7 TWAIN States, please see the TWAIN Specification for the final, official word.

TWAIN State 1: TWAIN Not Loaded

TWAIN State 2: Source Manager Loaded

TWAIN State 3: Source Manager Open

TWAIN State 4: DS Open

The Datasource is open but not enabled – You can talk to it, and it has not initiated any image acquisition. You can *negotiate* with the DS to determine the parameters of image acquisition. All of the functions listed under *Capability Negotiation* can be called in State 4 – The ones that *set* capabilities can only be safely called in State 4. You can *read* capabilities in State 4 or higher.

TWAIN State 5: DS Enabled

When the application enables the DS, it means 'go ahead with acquisition'. In this state, the DS will normally display a dialog or user interface, allowing the user to tweak the controls and set the settings, and eventually, perhaps, click on the button labeled Scan, Capture, or whatever.
If the DS is enabled *without user interface*, which most but not all DS's can do, then a proper DS will immediately begin acquiring an image, using the parameters negotiated in State 4.

TWAIN State 6: Transfer Ready

When an image is actually available and ready to transfer, the DS signals the application by posting a message, indicating that it is in State 6. In this state, the application must either cancel the transfer, or initiate transfer.

TWAIN State 7: Transfer In Progress

This is the State while image data is actively being transferred, and just after. The application is required to acknowledge the transfer, which moves the State to either 6 or 5, depending on whether another transfer is ready.

Appendix 1 - History

For specific changes between minor releases, please refer to the file History.txt, which is installed as part of the EZTwain Pro toolkit.

Changes from EZTwain Pro 3.0

Legal Changes

EZTwain Pro is now owned, sold and supported by Atalasoft, Inc. The license covering EZTwain Pro 4.0 is based on the license of other Atalasoft products, and differs in many important ways. Of the four or more license 'modalities' under which EZTwain Pro 3 was offered by Dosadi, only one, the Universal License, is offered by Atalasoft.

Technical Changes

The SDK installs into \Program Files\EZTwain4
or on 64-bit systems, into \Program Files (x86)\EZTwain4

Probably the biggest change, because it affects your deployment, troubleshooting, installer or setup, etc: The deployable binary files change names:

Old	New
Eztwain3.dll	Eztwain4.dll
EZCurl.dll	EZT4Curl.dll
EZDcx.dll	EZT4Dcx.dll
EZGif.dll	EZT4Gif.dll
EZJpeg.dll	EZT4Jpeg.dll
EZOcr.dll	EZT4Ocr.dll
EZPdf.dll	EZT4Pdf.dll
EZPng.dll	EZT4Png.dll
EZSymbol.dll	EZT4Symbol.dll
EZTiff.dll	EZT4Tiff.dll

The built-in barcode engine is now called the 'Native' engine instead of the 'Dosadi' engine, and the constant is renamed:

Old: EZBAR_ENGINE_DOSADI **New:** EZBAR_ENGINE_NATIVE

Functions marked as 'deprecated' in previous EZTwain 3.x releases have been removed. Quoting from History.txt:

Deleted Function	Suggestion
TWAIN_SetVendorKey	use TWAIN_UniversalLicense
TWAIN_OrganizationLicense	this license is no longer available
TWAIN_AcquireNative	use TWAIN_Acquire
TWAIN_WriteToFilename	use DIB_WriteToFilename
TWAIN_SaveToFilename	use DIB_WriteToFilename
TWAIN_WriteNativeToFile	use DIB_WriteToFilename
TWAIN_WriteNativeToFilename	use DIB_WriteToFilename
TWAIN_WriteDibToFile	use DIB_WriteToFilename
TWAIN_LoadNativeFromFilename	use DIB_LoadFromFilename
TWAIN_DibDepth, TWAIN_DibWidth	use DIB_Depth, DIB_Width
TWAIN_DibHeight	use DIB_Height
TWAIN_DibNumColors	no direct equivalent. DIB_ColorCount is related...
TWAIN_DibRowBytes	use DIB_RowBytes
TWAIN_DibReadRow	use DIB_ReadRow
TWAIN_CreateDibPalette	use DIB_CreatePalette
TWAIN_DrawDibToDC	use DIB_DrawToDC
TWAIN_NegotiatePixelTypes	closest is: TWAIN_SetPixelType
TWAIN_IsTransferReady	Redesign using TWAIN_IsDone

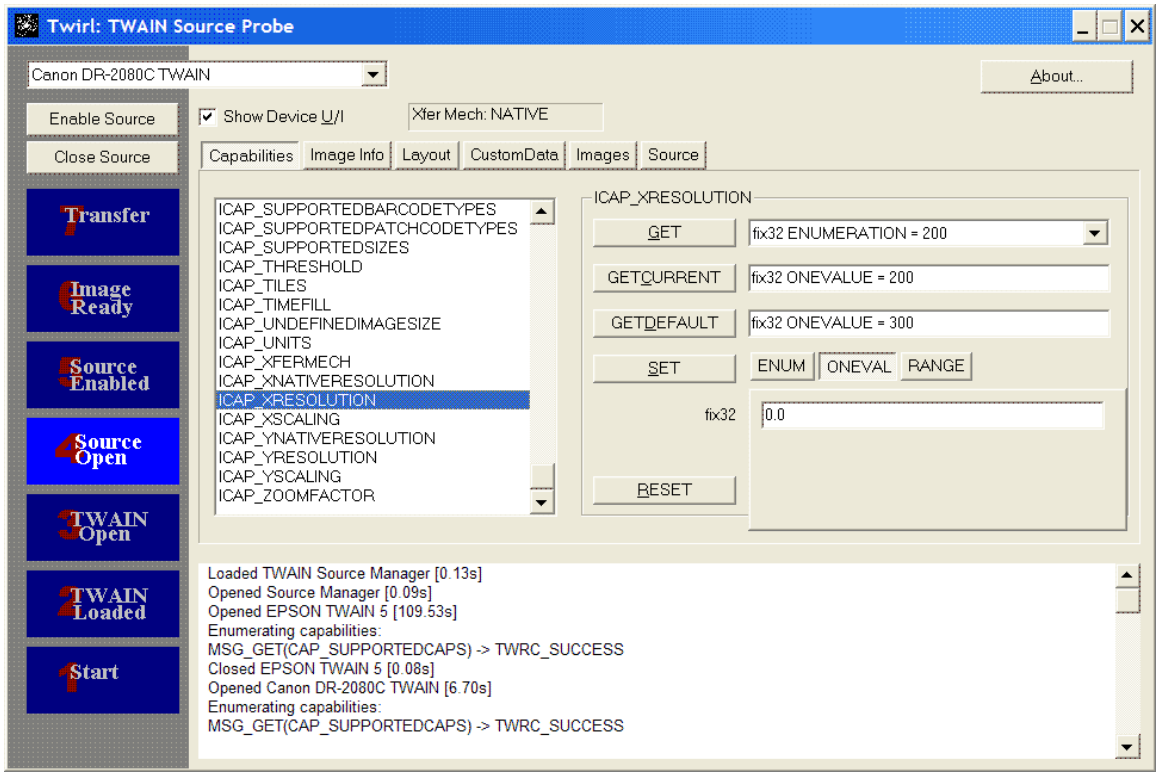
Appendix 2 - Working with Containers

Theory

Containers are the currency of TWAIN settings. TWAIN devices have dozens, sometimes hundreds of properties that can be queried and set: Is there paper in the feeder, resolution to use for the next scan, the serial number of the device. TWAIN calls these properties *capabilities*, and capability values move back and forth between application and TWAIN device in packages called *containers.* TWAIN calls this exchange *capability negotiation.*

Because it is necessary for the application and device to communicate not just the current value of a capability, but also its set of *possible* values, containers are quite rich and complex.

Advisory: There is no way to talk about containers in TWAIN without introducing a lot of terms and concepts. Before reading the following, we recommend you open our Twirl TWAIN Probe, included in the EZTwain Pro toolkit. It can examine any TWAIN device on your computer and display its capabilities. You can select any capability and Twirl will describe the containers that are returned by that capability. Seeing how containers are actually used by your specific TWAIN device may make the following more comprehensible.



Containers come in four flavors, depending on what they need to represent:

A **OneValue** container holds a single value, like: 200. Commonly used to select a specific value for a capability, such as 200 DPI for resolution, or to answer a simple query like CAP_FEEDERLOADED ("is there paper in the feeder".)

An **Enumeration** container is a set of values, with two values called out – Current and Default. When queried with TWAIN_Get, many TWAIN capabilities return an Enumeration representing the set of valid values, plus the current value and the default (reset) value.

A **Range** container describes a set of values by giving a minimum value, a maximum value, and an increment or *step.* It's like a for-loop. Like an Enumeration, the Range container can specify a Current value and a Default value.

An **Array** container holds a list of values, none of which are special. Array containers are used with capabilities whose values are actually sets - like CAP_SUPPORTEDCAPS.

All the items in a container are of the same basic *item type*, chosen from the following unecessarily baroque set.

TWAIN Container Item Types

Item Type	What each item holds
TWTY_INT8	8-bit signed integer (-128..127)
TWTY_INT16	16-bit signed integer (-32768..32767)
TWTY_INT32	32-bit signed integer (go figure)
TWTY_UINT8	8-bit unsigned integer (0..255)
TWTY_UINT16	16-bit unsigned integer (0..65535)
TWTY_UINT32	32-bit unsigned integer (0.. 4294967295)
TWTY_BOOL	TRUE (1) or FALSE (0)
TWTY_FIX32	32-bit fractional number about -32767.9999 to 32767.9999
TWTY_FRAME	a rectangle defined by 4 TWTY_FIX32 values
TWTY_STR32	32-character ANSI string
TWTY_STR64	64-character ANSI string
TWTY_STR128	128-character ANSI string
TWTY_STR255	255-character ANSI string
TWTY_STR1024	1024-character ANSI string
TWTY_UNI512	512-character UNICODE string

All these appear as defined constants in the EZTwain declaration file for your programming language. EZTwain tries to hide the details from you, but you should be aware that underneath, every container has a specific item type, and every capability has a specific item type that it works with.

Practice

EZTwain represents containers with a handle called an HCONTAINER. This is basically an unsigned integer.

Note: Because these are handles and not 'objects', you are responsible for releasing them when you are done with them, using CONTAINER_Free. The amount of

memory involved is not as large as with images, but it is good practice to avoid leaks.

The function TWAIN_Set (EZTwain.SetCap) is the fundamental function for sending a container to a capability. This function can only be called in TWAIN State 4, and will produce an error if called in any other state. TWAIN State 4 is the capability-setting state.

TWAIN_Get (EZTwain.GetCap) is the core function for querying the value of a capability. The TWAIN State must be 4 or higher, or this function will produce an error. TWAIN_GetCurrent (EZTwain.GetCurrent) is used somewhat less often - it is supposed to return just the current value of a capability, but *many* TWAIN devices return the same container for both TWAIN_Get and TWAIN_GetCurrent.

Example 1. Enumerate Resolutions

Suppose you want to find out what resolutions a device supports. Assume the device is open from a previous call to TWAIN_OpenSource or TWAIN_OpenDefaultSource, so the device is in State 4. This is C/C++ code, the comments should help you translate to other languages:

```
// declare hcon as a container handle
HCONTAINER hcon;
// Set the unit of measure to inches, because we want
// resolution in samples per inch.  In theory if current units
// were cm, resolution would be returned in samples per cm!
// (Don't count on that though, some devices always use DPI.)
TWAIN_SetUnits(TWUN_INCHES);
// We will use TWAIN_Get (EZTwain.GetCap) to get the
// x-resolution capability value as a container.
// You can assume that this container will list all allowed
// (X) resolutions, plus the current and default resolution.
hcon = TWAIN_Get(ICAP_XRESOLUTION);
// Notice that TWAIN distinguishes X-resolution from Y-
resolution,
// although they are normally the same sets of values, and
// *usually* setting one will set the other to the same value.
if (hcon != 0) {
    // It is mandatory for every TWAIN driver to support
    // ICAP_XRESOLUTION and ICAP_YRESOLUTION, so in this case
    // the check for valid hcon is pretty silly.  If this
    // capability was optional, hcon = 0 would mean that the
    // capability is not supported.
    // Declare floating-point variables for resolution values:
    double dXRes, dXResCurrent, dXResDefault;
    dXResCurrent = CONTAINER_CurrentValue(hcon);
    dXResDefault = CONTAINER_DefaultValue(hcon);
    // Loop through all the supported values.
    // CONTAINER_ItemCount is always the number of values in hcon
    for (int i = 0; i < CONTAINER_ItemCount(hcon); i++) {
        // get the ith item in hcon, item 0 is the first:
        dXRes = CONTAINER_FloatValue(hcon, i);
        // do something with dXRes, like add to a listbox.
    } // end for
    CONTAINER_Free(hcon);        // release the container
}
// Just for fun, reset X-resolution to default value:
TWAIN_Reset(ICAP_XRESOLUTION);
```

Example 2. Custom Capability

For simple query/set situations, you may not need to use containers at all - you can usually use ETwain's higher-level capability functions.

TWAIN reserves the upper 32,768 capability codes for *custom capabilities* - capabilities that are defined by the device vendor for a specific product or product family. There are only two ways to find out what these capabilities do - Ask the manufacturer, or experiment - for example, with Twirl.

The Canon DR-2080C has a custom capability code hex 8025 (decimal 32805). Using Twirl we find that this capability uses type TWTY_INT32, and it returns an enumeration containing the values 2 and 0. By setting this capability through Twirl and enabling the device so its dialog pops up, we find that this capability controls the

Automatic Border Removal feature: 2=enabled, 0=disabled. This is enough information for us to control this capability in our code, using the function TWAIN_SetCapability, which is specifically designed for setting custom capabilities:

```
// Try to enable Automatic Border Removal on DR-2080C:
TWAIN_SetCapability(32805, 2);
```

Here is what this function is doing for you at a lower level:

```
// Declare a variable to hold a container handle
HCONTAINER hcon
// Create a one-value container of the correct type:
hcon = CONTAINER_OneValue(TWTY_INT32, 2);
// Try to set the Automatic Border Removal capability:
TWAIN_Set(32805, hcon);
// Don't leak memory:
CONTAINER_Free(hcon);
```

To read the current value of this capability, you could use code like this:

```
// Declare a variable to hold a container handle (32-bit integer)
HCONTAINER hcon
// Get the container for the Automatic Border Removal capability:
hcon = TWAIN_Get(32805);
if (hcon != 0) {
    // valid container handle returned.
    // Test the current value represented by this container.
    // If it's a Range or Enumeration, find the Current value
    // If it's a OneValue, just use that (one) value:
    if (CONTAINER_CurrentValue(hcon) == 2) {
        ABRenabled = TRUE;
    } else {
        ABRenabled = FALSE;
    }
    CONTAINER_Free(hcon)
}
```

Example 3. Endorser/Imprinter

Some scanners have an internal printer (called an *imprinter* or *endorser*) that can print something on each scanned page. There is wide variation in when, where, and what they print - before or after scanning, top, bottom, front, back, strings, serial numbers, etc. The TWAIN standard draws the following distinction:

> "Imprinters are used to print data on documents at the time of scanning, and may be used for any purpose. Endorsers are more specific in nature, stamping some kind of proof of scanning on the document." TWAIN 1.9 p. 9-393

EZTwain *does not include any functions specifically for controlling an imprinter or endorser*, so you must roll your own from lower-level functions.

```
HCONTAINER hcon;
// Get the list of available printer devices:
hcon = TWAIN_Get(CAP_PRINTER);
if (hcon == 0) return;           // no printer available
// Enumerate the available printer types:
for (int i = 0; i < CONTAINER_ItemCount(hcon); i++) {
    // Look at each entry in the container, the TWAIN spec lists
    // the printer types under CAP_PRINTER:
    int PrinterType = CONTAINER_IntValue(hcon, i);
} // end for
CONTAINER_Free(hcon);
```

Here is code to enable the current printer (whatever type it is), and try to tell it to imprint on each page a string of the form Document-NNN-eztwain, with NNN starting at 640. Your scanner may not support this mode, of course.

```
// Enable the printer:
TWAIN_SetCapBool(CAP_PRINTERENABLED, TRUE);
// Tell printer to print a 'compound string' (mode 2),
// meaning <prefix-string><number><suffix-string>
TWAIN_SetCapOneValue(CAP_PRINTERMODE, TWTY_UINT16, 2);
// Tell printer to start serial numbers at 640:
TWAIN_SetCapOneValue(CAP_PRINTERINDEX, TWTY_UINT32, 640);
// Now, create a onevalue string (STR255) container, which
// can be used to set both the prefix and suffix:
hcon = CONTAINER_OneValue(TWTY_STR255, 0);
// First set "Document-" as the container's one value:
CONTAINER_SetItemString(hcon, 0, "Document-");
// Use that to set the (prefix?) printer string:
TWAIN_Set(CAP_PRINTERSTRING, hcon);
// The container still exists, change its value
// to "-eztwain", our suffix
CONTAINER_SetItemString(hcon, 0, "-eztwain");
// Set the printer's suffix string:
TWAIN_Set(CAP_PRINTERSUFFIX, hcon);
// Done with this container, don't leak memory.
CONTAINER_Free(hcon);
```

Appendix 3 - Multithreading with EZTwain

It is possible to use EZTwain in a multithreaded program: Several customers have done this. You can follow the *Simple Rule* or the *Complicated Rules*.

Simple Rule for Multithreading

Make all EZTwain calls in one single thread, and if you pass a Window handle to EZTwain, it must belong to (have been created by) that thread.

Complicated Rules for Multithreading

1. Once a call is made in thread T that moves the TWAIN State above State 2, all calls that require a State above State 2 must be made in thread T until the state drops to State 2 or lower. This includes all calls that change, or can change, the State, and all calls that set or query scanning capabilities or parameters. This includes: TWAIN_GetSourceList and TWAIN_(Get)NextSourceName.

2. Any window handle passed as a parameter to an EZTwain call from thread T must have been created in thread T.

3. DIB_ functions can be called in any thread, as long as they are given a valid DIB handle. This also applies to:
   TWAIN_BeginMultipageFile, TWAIN_DibWritePage, TWAIN_EndMultipageFile
   TWAIN_MultipageCount
   TWAIN_FormatOfFile.

4. CONTAINER_ functions can be called in any thread, as long as they are given a valid HCONTAINER.

5. Certain EZTwain global-state functions can be called in any thread, but EZTwain does not protect itself from race conditions. For example:
   TWAIN_State
   TWAIN_RegisterApp, TWAIN_SetAppTitle
   TWAIN_SetApplicationKey, TWAIN_SetVendorKey
   TWAIN_ApplicationLicense, TWAIN_OrganizationLicense
   TWAIN_SetHideUI / TWAIN_GetHideUI
   TWAIN_SetMultiTransfer / TWAIN_GetMultiTransfer
   TWAIN_SuppressErrorMessages
   TWAIN_IsMultipageAvailable, TWAIN_IsJpegAvailable, etc.
   TWAIN_FormatFromExtension
   TWAIN_SetFileAppendFlag / TWAIN_GetFileAppendFlag
   TWAIN_SetSaveFormat / TWAIN_GetSaveFormat
   TWAIN_SetMultipageFormat / TWAIN_GetMultipageFormat
   TWAIN_SetJpegQuality / TWAIN_GetJpegQuality
   TWAIN_SetTiff* / TWAIN_GetTiff*
   TWAIN_EasyVersion
   TWAIN_DisableParent / TWAIN_GetDisableParent

Appendix 4 - EZTwain Datatypes

EZTwain Datatypes

C/C++ type	physical representation
void	used as a placeholder to mean 'nothing' or 'no value'
int	signed word (32 bits)
unsigned	unsigned word (32 bits)
double	64-bit (8-byte) IEEE binary floating number
double* (1)	pointer to a *double*.
LPSTR	ointer to 0-terminated string of 8-bit ANSI characters.
char*	same as LPSTR
LPCSTR	same as LPSTR but pointed-to string cannot be modified.
LPVOID	pointer, not specified to what.
BYTE*	pointer to a byte (usually used to point to a buffer of data.)
LPMSG	pointer to a Windows MSG structure.
HANDLE	unsigned number, used to designate an object.
HWND	HANDLE of a window.
HCONTAINER	HANDLE to an EZTwain container object.
BOOL	32-bit word either 0 (FALSE) or 1 (TRUE)
HPALETTE	HANDLE specifically for a Windows GDI palette.
HDC	HANDLE of a Windows GDI Device Context.
HFILE	HANDLE of a Windows file, for CreateFile, Write, _lclose, …

Note 1: In EZTwain, parameters declared as double* are used by the called function to return values. In Visual Basic these would be declared as ByRef parameters. If your language has the concept of *reference parameter*, you would translate *double** as a parameter of type *double passed by reference.*

Index