TWAIN Specification

Release 1.6

This document has been ratified by the TWAIN Working Group Committee as of February 5, 1996

Notice

The Working Group grants customer ("Customer") the worldwide, royalty-free, non-exclusive license to reproduce and distribute the software and documentation of the TWAIN toolkit ("TWAIN Toolkit"). The TWAIN Toolkit was designed to be used by third parties to assist them in becoming compliant with the TWAIN standard, but it has not been developed to the standards of a commercial product. Consequently, the TWAIN Toolkit is provided AS IS without any warranty. THE WORKING GROUP DISCLAIMS ALL WARRANTIES IN THE TWAIN TOOLKIT WHETHER IMPLIED, EXPRESS OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT OF THIRD PARTY RIGHTS AND FITNESS FOR A PARTICULAR PURPOSE. The Working Group disclaims all liability for damages, whether direct, indirect, special, incidental, or consequential, arising from the reproduction, distribution, modification or other use of the TWAIN Toolkit.

As a condition of this license, Customer agrees to include in software programs based in whole or in part on the TWAIN Toolkit the following provisions in (i) the header or similar file in such software and (ii) prominently in its documentation and to require its sublicensees to include these provisions in similar locations: THE TWAIN TOOLKIT IS DISTRIBUTED AS IS. THE DEVELOPER AND THE DISTRIBUTORS OF THE TWAIN TOOLKIT EXPRESSLY DISCLAIM ALL IMPLIED, EXPRESS OR STATUTORY WARRANTIES INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT OF THIRD PARTY RIGHTS AND FITNESS FOR A PARTICULAR PURPOSE. NEITHER THE DEVELOPERS NOR THE DISTRIBUTORS WILL BE LIABLE FOR DAMAGES, WHETHER DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL, AS A RESULT OF THE REPRODUCTION, MODIFICATION, DISTRIBUTION OR OTHER USE OF THE TWAIN TOOLKIT.

Trademarks

Adobe is a registered trademark of Adobe Systems, Inc.
Macintosh is a registered trademark of Apple Computer, Inc.
Microsoft is a registered trademark of Microsoft Corporation.
Windows is a trademark of Microsoft Corporation.

Printing History

Edition 1	February, 1992 (TWAIN Version 1.0)
Edition 1	May, 1993 (TWAIN Version 1.5)
Edition 2	February, 1996 (TWAIN Version 1.6)

100-15008

Acknowledgments

The TWAIN Working Group acknowledges many people and their companies for their contributions to the 1.6 specification. Their hard work in editing, proofreading and discussing the changes have been invaluable. Among those we especially want to thank are:

Logitech

Gil Regev, Spike McLarty and Poul Hansen

Hewlett-Packard

Elen Hunt, Teresa Simske, Bill DeVoe, Kevin Bier and Mark Seaman

Documagix

Angie Philips and Sam Hahn

Microtek Lab

Craig Lindley and Julia Hsieh

Nikon Electronic Imaging

Khouri Giordano

Canon

Adam Wang

View Software

Rakesh Agarwal

We also thank the TWAIN community at large for their opinions and contributions to the specification.

1

Introduction

Chapter Contents

The Need for Consistency

With the introduction of scanners and other image acquisition devices, users eagerly discovered the value of incorporating images into their documents and other work. However, supporting the display and manipulation of this raster data placed a high cost on application developers. They needed to create user interfaces and build in device control for the wide assortment of available image devices. Once their application was prepared to support a given device, they faced the discouraging reality that devices continue to be upgraded with new capabilities and features. Application developers found themselves continually revising their product to stay current.

Developers of both the image acquisition devices and the software applications recognized the need for a standard communication between the image devices and the applications. A standard would benefit both groups as well as the users of their products. It would allow the device vendors' products to be accessed by more applications and application vendors could access data from those devices without concern for which type of device, or particular device, provided it. TWAIN was developed because of this need for consistency and simplification.

The Elements of TWAIN

TWAIN defines a standard software protocol and API (application programming interface) for communication between software applications and image acquisition devices (the source of the data).

The three key elements in TWAIN are:

- **The application software** - An application must be modified to use TWAIN.
- **The Source Manager software** - This software manages the interactions between the application and the Source. This code is provided in the TWAIN Developer's Toolkit and should be shipped for free with each TWAIN application and Source.
- **The Source software** - This software controls the image acquisition device and is written by the device developer to comply with TWAIN specifications. Traditional device drivers are now included with the Source software and do not need to be shipped by applications.

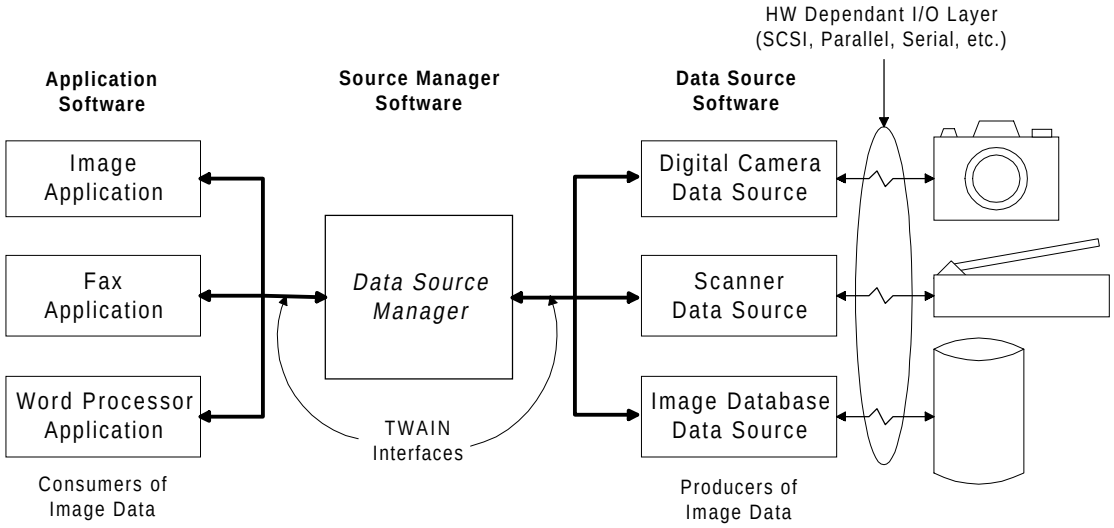


Figure 1-1. TWAIN Elements

The Benefits of Using TWAIN

For the Application Developer

- Allows you to offer users of your application a simple way to incorporate images from any compatible raster device without leaving your application.
- Saves time and dollars. If you currently provide low-level device drivers for scanners, etc., you no longer need to write, support, or ship these drivers. The TWAIN-compliant image acquisition devices will provide Source software modules that eliminate the need for you to create and ship device drivers.
- Permits your application to access data from any TWAIN-compliant image peripheral simply by modifying your application code once using the high-level TWAIN application programming interface. No customization by product is necessary. TWAIN image peripherals can include desktop scanners, hand scanners, digital cameras, frame grabbers, image databases, or any other raster image source that complies to the TWAIN protocol and API .
- Allows you to determine the features and capabilities that an image acquisition device can provide. Your application can then restrict the Source to offer only those capabilities that are compatible with your application's needs and abilities.
- Eliminates the need for your application to provide a user interface to control the image acquisition process. There is a software user interface module shipped with every TWAIN-compliant Source device to handle that process. Of course, you may provide your own user interface for acquisition, if desired.

For the Source Developer

- Increases the use and support of your product. More applications will become image consumers as a result of the ease of implementation and breadth of device integration that TWAIN provides.
- Allows you to provide a proprietary user interface for your device. This lets you present the newest features to the user without waiting for the applications to incorporate them into their interfaces.
- Saves money by reducing your implementation costs. Rather than create and support various versions of your device control software to integrate with various applications, you create just a single TWAIN-compliant Source.

For the End User

- Gives users a simple way to incorporate images into their documents. They can access the image in fewer steps because they never need to leave your application.

The Creation of TWAIN

TWAIN was created by a small group of software and hardware companies in response to the need for a proposed specification for the imaging industry. The Working Group's goal was to provide an open, multi-platform solution to interconnect the needs of raster input devices with application software. The original Working Group was comprised of representatives from five companies: Aldus, Caere, Eastman Kodak, Hewlett-Packard, and Logitech. Three other companies, Adobe, Howtek, and Software Architects also contributed significantly.

The design of TWAIN began in January, 1991. Review of the original TWAIN Developer's Toolkit occurred from April, 1991 through January, 1992. The original Toolkit was reviewed by the TWAIN Coalition. The Coalition includes approximately 300 individuals representing 200 companies who continue to influence and guide the future direction of TWAIN.

During the creation of TWAIN, the following architecture objectives were adhered to:

- **Ease of Adoption** - Allow an application vendor to make their application TWAIN-compliant with a reasonable amount of development and testing effort. The basic features of TWAIN should be implemented just by making modest changes to the application. To take advantage of a more complete set of functionality and control capabilities, more development effort should be anticipated.
- **Extensibility** - The architecture must include the flexibility to embrace multiple windowing environments spanning various host platforms (Macintosh, MS Windows, Motif, etc.) and facilitate the exchange of various data types between Source devices and destination applications. Currently, only the raster image data type is supported but suggestions for future extensions include text, facsimile, vector graphics, and others.
- **Integration** - Key elements of the TWAIN implementation "belong" in the operating system. Migration of these elements from TWAIN into the operating system is proposed. TWAIN must be implemented to encourage backward compatibility (extensibility) and smooth migration into the operating system. An implementation that minimizes the use of platform-specific mechanisms will have enhanced longevity and adoptability.
- **Easy Application <-> Source Interconnect** - A straight-forward Source identification and selection mechanism will be supplied. The application will drive this mechanism through a simple API. This mechanism will also establish the data and control links between the application and Source. It will support capability and configuration communication and negotiation between the application and Source.
- **Encapsulated Human Interface** - A device-native user interface will be required in each Source. The application can optionally override this native user interface while still using the Source to control the physical device.

The Systems Supported by TWAIN

This version of TWAIN (Release v1.6) provides Source Manager software for use by software applications operating under the Apple® Macintosh® Operating System (System 6.x or 7.x) or Microsoft® Windows™ (Version 3.x, Win 95 and NT).

The Contents of the Toolkit

The TWAIN Developer's Toolkit provides hardware and software developers with three important components:

- The description and specification information needed to learn about the TWAIN protocol and API. Every attempt has been made to make this document accurate. However, be aware that Technical Notes may be added to the TWAIN Toolkit to correct errors or clarify areas of confusion. Technical Notes are posted to CompuServe and AppleLink when developed. Please check these services occasionally to determine if there is any information that applies to you.
- Sample code showing applications and Sources. This includes a Sample Application that is used to exercise sources. Documentation of the sample source and application is included in the toolkit.
- Source Manager modules to coordinate communications between applications and Sources.

The Source Manager is provided on the Developer's Disks. There is a separate Source Manager file for each platform on which TWAIN has been implemented (currently Macintosh and Windows). TWAIN requests that you ship a copy of the Source Manager file with each of your TWAIN-compliant products. This policy ensures that your users will have a copy of the Source Manager. The Source Manager is relatively small in size (less than 90 Kbytes) and is made available to developers free of charge.

The sample Source code programs and the Source Manager module are available to any developer free of license fee. On-line technical support is available from any of the Working Group companies. Refer to Chapter 11 for more information.

Every attempt has been made to make this document accurate. However, if there are any discrepancies between this document and the contents of the file called TWAIN.H, the TWAIN.H file is considered correct.

2

Technical Overview

The TWAIN protocol and API are easiest to understand when you see the overall picture. This chapter describes:

Chapter Contents

The TWAIN Architecture

The transfer of data is made possible by three software elements that work together in TWAIN: the application, the Source Manager, and the Source.

These elements use the architecture of TWAIN to communicate. The TWAIN architecture consists of four layers:

- Application
- Protocol
- Acquisition
- Device

The TWAIN software elements occupy the layers as illustrated below. Each layer is described in the sections that follow.

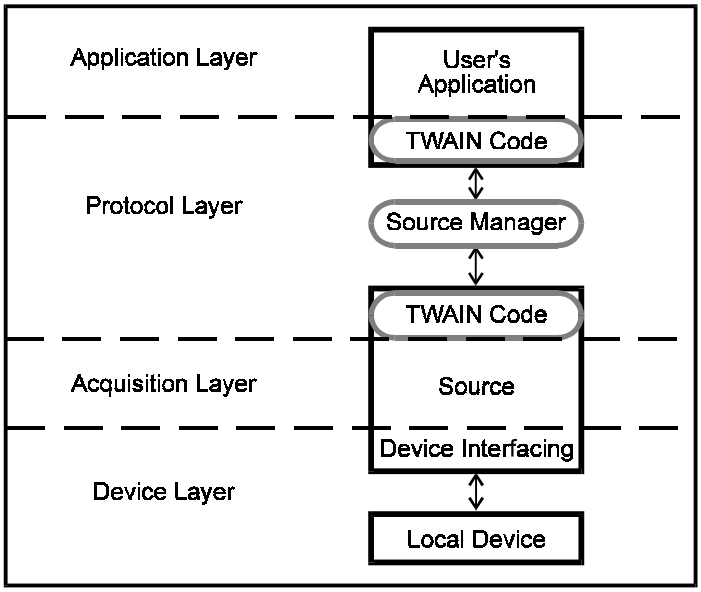


Figure 2-1. TWAIN Software Elements

Application

The user's software application executes in this layer.

TWAIN describes user interface guidelines for the application developer regarding how users access TWAIN functionality and how a particular Source is selected.

TWAIN is not concerned with how the application is implemented. TWAIN has no effect on any inter-application communication scheme that the application may use.

Protocol

The protocol is the "language" spoken and syntax used by TWAIN. It implements precise instructions and communications required for the transfer of data.

The protocol layer includes:

- The portion of application software that provides the interface between the application and TWAIN
- The TWAIN Source Manager provided by TWAIN
- The software included with the Source device to receive instructions from the Source Manager and transfer back data and Return Codes

The contents of the protocol layer are discussed in more detail in a following section called "Communication between the Elements of TWAIN."

Acquisition

Acquisition devices may be physical (like a scanner) or logical (like an image database). The software elements written to control acquisitions are called Sources and reside primarily in this layer.

The Source transfers data for the application. It uses the format and transfer mechanism agreed upon by the Source and application.

The Source always provides a built-in user interface that controls the device(s) the Source was written to drive. An application can override this and present its own user interface for acquisition, if desired.

Device

This is the location of traditional low-level device drivers. They convert device-specific commands into hardware commands and actions specific to the particular device the driver was written to accompany. Applications that use TWAIN no longer need to ship device drivers because they are part of the Source.

TWAIN is not concerned with the device layer at all. The Source hides the device layer from the application. The Source provides the translation from TWAIN operations and interactions with the Source's user interface into the equivalent commands for the device driver that cause the device to behave as desired.

Note: The Protocol layer is the most thoroughly and rigidly defined to allow precise communications between applications and Sources. The information in this document concentrates on the Protocol and Acquisition layers.

The User Interface to TWAIN

When an application uses TWAIN to acquire data, the acquisition process may be visible to the application's users in the following three areas:

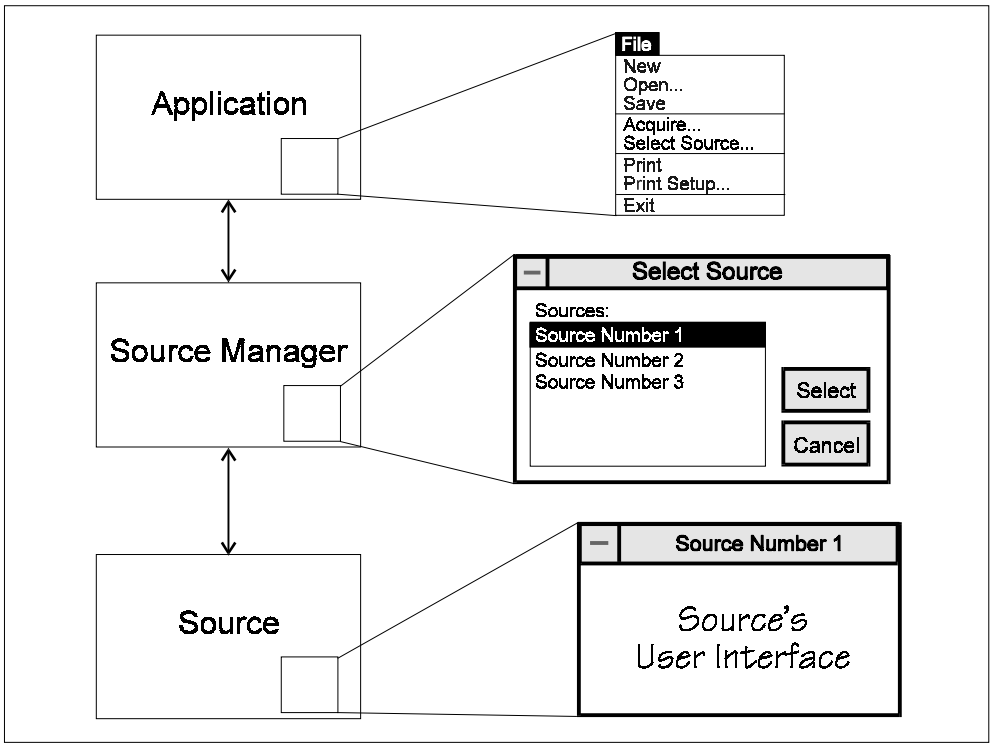


Figure 2-2. Data Acquisition Process

The Application

The user needs to select the device from which they intend to acquire the data. They also need to signal when they are ready to have the data transferred.

To allow this, TWAIN strongly recommends the application developer add two options to their File menu:

- **Select Source** - to select the device
- **Acquire** - to begin the transfer process

The Source Manager

When the user chooses the Select Source option, the application requests that the Source Manager display its **Select Source dialog box**. This lists all available devices and allows the user to highlight and select one device. If desired, the application can write its own version of this interface.

The Source

Every TWAIN-compliant Source provides a user interface specific to its particular device. When the application user selects the Acquire option, the **Source's User Interface** may be displayed. If desired, the application can write its own version of this interface, too.

Communication Between the Elements of TWAIN

Communication between elements of TWAIN is possible through two entry points. They are called DSM_Entry( ) and DS_Entry( ). DSM means Data Source Manager and DS means Data Source.

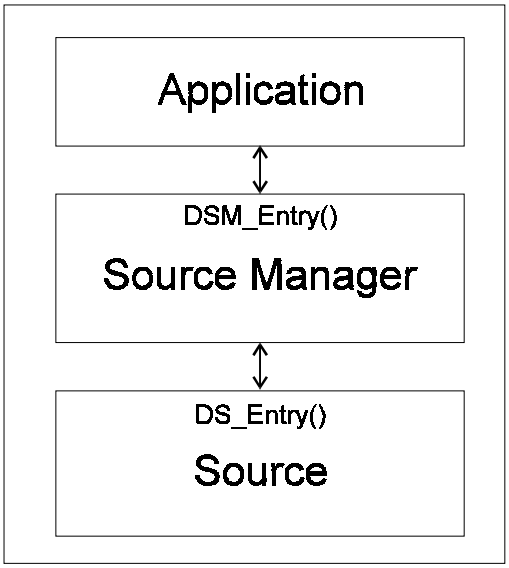


Figure 2-3. Entry Points for Communicating Between Elements

The Application

The goal of the application is to acquire data from a Source. However, applications cannot contact the Source directly. All requests for data, capability information, error information, etc. must be handled through the Source Manager.

Approximately fifty operations are defined by TWAIN. The application sends them to the Source Manager for transmission. The application specifies which element, Source Manager or Source, is the final destination for each requested operation.

The application communicates to the Source Manager through the Source Manager's only entry point, the **DSM_Entry( )** function.

The parameter list of the DSM_Entry function contains:

- An identifier structure providing information about the application that originated the function call
- The destination of this request (Source Manager or Source)
- A triplet that describes the requested operation. The triplet specifies:
  - ✓ Data Group for the Operation (DG_)
  - ✓ Data Argument Type for the Operation (DAT_)
  - ✓ Message for the Operation (MSG_)

  (These are described more in the section called "The Use of Operation Triplets" located later in this chapter.)
- A pointer field to allow the transfer of data

The function call returns a value (the Return Code) indicating the success or failure of the operation.

Written in C code form, the function call looks like this:

On Windows

```
TW_UINT16 FAR PASCAL DSM_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       pTW_IDENTITY   pDest,      // destination of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

On Macintosh

```
FAR PASCAL TW_UINT16 DSM_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       pTW_IDENTITY   pDest,      // destination of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

Note: Data type definitions are covered in Chapter 8 of this document and in the file called TWAIN.H which is shipped on the developer's disk.

The Source Manager

The Source Manager provides the communication path between the application and the Source, supports the user's selection of a Source, and loads the Source for access by the application. Communications from application to Source Manager arrive in the DSM_Entry( ) entry point.

- **If the destination in the DSM_Entry call is the Source Manager** - The Source Manager processes the operation itself.
- **If the destination in the DSM_Entry call is the Source** - The Source Manager translates the parameter list of information, removes the destination parameter and calls the appropriate Source. To reach the Source, the Source Manager calls the Source's **DS_Entry( )** function. TWAIN requires each Source to have this entry point.

Written in C code form, the DS_Entry function call looks like this:

On Windows

```
TW_UINT16 FAR PASCAL DS_Entry
      (pTW_IDENTITY   pOrigin,    // source of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
      );
```

On Macintosh

```
FAR PASCAL TW_UINT16 DS_Entry
      (pTW_IDENTITY   pOrigin,    // source of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
      );
```

In addition, the Source Manager can initiate three operations that were not originated by the application. These operation triplets exist just for Source Manager to Source communications and are executed by the Source Manager while it is displaying its Select Source dialog box. The operations are used to identify the available Sources and to open or close Sources.

The implementation of the Source Manager differs between the supported systems:

On Windows

The Source Manager for Windows is a Dynamic Link Library (DLL).

The Source Manager can manage simultaneous sessions between many applications with many Sources. That is, the same instance of the Source Manager is shared by multiple applications.

On Macintosh

The Source Manager for Macintosh is a code resource.

Each application gets a private copy of the Source Manager and the Source(s) it opens. The separate instances of the Source Manager and Sources can coordinate among themselves.

The Source

The Source receives operations either from the application, via the Source Manager, or directly from the Source Manager. It processes the request and returns the appropriate Return Code (the codes are prefixed with TWRC_) indicating the results of the operation to the Source Manager. This Return Code is then passed back to the application as the return value of its DSM_Entry( ) function call. If the operation was unsuccessful, a Condition Code (the codes are prefixed with TWCC_) containing more specific information is set by the Source. Although the Condition Code is set, it is not automatically passed back. The application must invoke an operation to inquire about the contents of the Condition Code.

The implementation of the Source is the same as the implementation of the Source Manager:

On Windows

The Source is a Dynamic Link Library (DLL) so applications share the same copy of each element.

On Macintosh

The Source is implemented as a Code Resource.

Communication Flowing from Source to Application

The majority of operation requests are initiated by the application and flow to the Source Manager and Source. The Source, via the Source Manager, is able to pass back data and Return Codes.

However, there are two times when the Source needs to interrupt the application and request that an action occur. These situations are:

- **Notify the application that a data transfer is ready to occur.** The time required for a Source to prepare data for a transfer will vary. Rather than have the application wait for the preparation to be complete, the Source just notifies it when everything is ready. The MSG_XFERREADY notice is used for this purpose.
- **Request that the Source's user interface be disabled.** This notification should be sent by the Source to the application when the user clicks on the "Close" button of the Source's user interface. The MSG_CLOSEDSREQ notice is used for this purpose.

These notices are presented to the application in its event (or message) loop. The process used for these notifications is covered more fully in Chapter 3 in the discussion of the application's event loop.

The Use of Operation Triplets

The DSM_Entry( ) and DS_Entry( ) functions are used to communicate operations. An operation is an action that the application or Source Manager invokes. Typically, but not always, it involves using data or modifying data that is indicated by the last parameter (pData) in the function call.

Requests for actions occur in one of these ways:

From	To	Using this function
The application	The Source Manager	DSM_Entry with the pDest parameter set to NULL
The application	The Source (via the Source Manager)	DSM_Entry with the pDest parameter set to point to a valid structure that identifies the Source
The Source Manager	The Source	DS_Entry

The desired action is defined by an operation triplet passed as three parameters in the function call. Each triplet uniquely, and without ambiguity, specifies a particular action. No operation is specified by more than a single triplet. The three parameters that make up the triplet are Data Group, Data Argument Type, and Message ID. Each parameter conveys specific information.

Data Group (DG_xxxx)

Operations are divided into large categories by the Data Group identifier. There are currently only two defined in TWAIN:

- **CONTROL** (The identifier is DG_CONTROL.): These operations involve control of the TWAIN session. An example where DG_CONTROL is used as the Data Group identifier is the operation to open the Source Manager.
- **IMAGE** (The identifier is DG_IMAGE.): These operations work with image data. Image is the only type of data currently defined by TWAIN. This can be expanded later and Sources can, in theory, supply more than one data type. An example where DG_IMAGE is used as the Data Group is an operation that requests the transfer of data.

Data Argument Type (DAT_xxxx)

This parameter of the triplet identifies the type of data that is being passed or operated upon. The argument type may reference a data structure or a variable. There are many data argument types. One example is DAT_IDENTITY.

The DAT_IDENTITY type is used to identify a TWAIN element such as a Source. Remember, from the earlier code example, data is typically passed or modified through the pData parameter of the DSM_Entry and DS_Entry. In this case, the pData parameter would point to a data structure of type TW_IDENTITY. Notice that the data argument type begins with DAT_xxx and the associated data structure begins with TW_xxx and duplicates the second part of the name. This pattern is followed consistently for most data argument types and their data structures. Any exceptions are noted on the reference pages in Chapters 7 and 8.

Message ID (MSG_xxxx)

This parameter identifies the action that the application or Source Manager wishes to have taken. There are many different messages such as MSG_GET or MSG_SET. They all begin with the prefix of MSG_.

Here are three examples of operation triplets:

The triplet the application sends to the Source Manager to open the Source Manager module is:

DG_CONTROL / DAT_PARENT / MSG_OPENDSM

The triplet that the application sends to instruct the Source Manager to display its Select Source dialog box and thus allow the user to select which Source they plan to obtain data from is:

DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT

The triplet the application sends to transfer data from the Source into a file is:

DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET

The State-Based Protocol

The application, Source Manager, and Source must communicate to manage the acquisition of data. It is logical that this process must occur in a particular sequence. For example, the application cannot successfully request the transfer of data from a Source before the Source Manager is loaded and prepared to communicate the request.

To ensure the sequence is executed correctly, the TWAIN protocol defines seven states that exist in TWAIN sessions. A session is the period while an application is connected to a particular Source via the Source Manager. The period while the application is connected to the Source Manager is another unique session. At a given point in a session, the TWAIN elements of Source Manager and Source each occupy a particular state. Transitions to a new state are caused by operations requested by the application or Source. Transitions can be in the forward or backward direction. Most transitions are single-state transitions. For example, an operation moves the Source Manager from State 1 to State 2 not from State 1 to State 3. (There are situations where a two-state transition may occur. They are discussed in Chapter 3.)

When viewing the state-based protocol, it is helpful to remember:

States 1, 2, and 3

- Are occupied only by the Source Manager.
- The Source Manager never occupies a state greater than State 3.

States 4, 5, 6, and 7

- Are occupied exclusively by Sources.
- A Source never has a state less than 4 if it is open. If it is closed, it has no state.
- If an application uses multiple Sources, each connection is a separate session and each open Source “resides” in its own state without regard for what state the other Sources are in.

The State Transition Diagram looks like this:

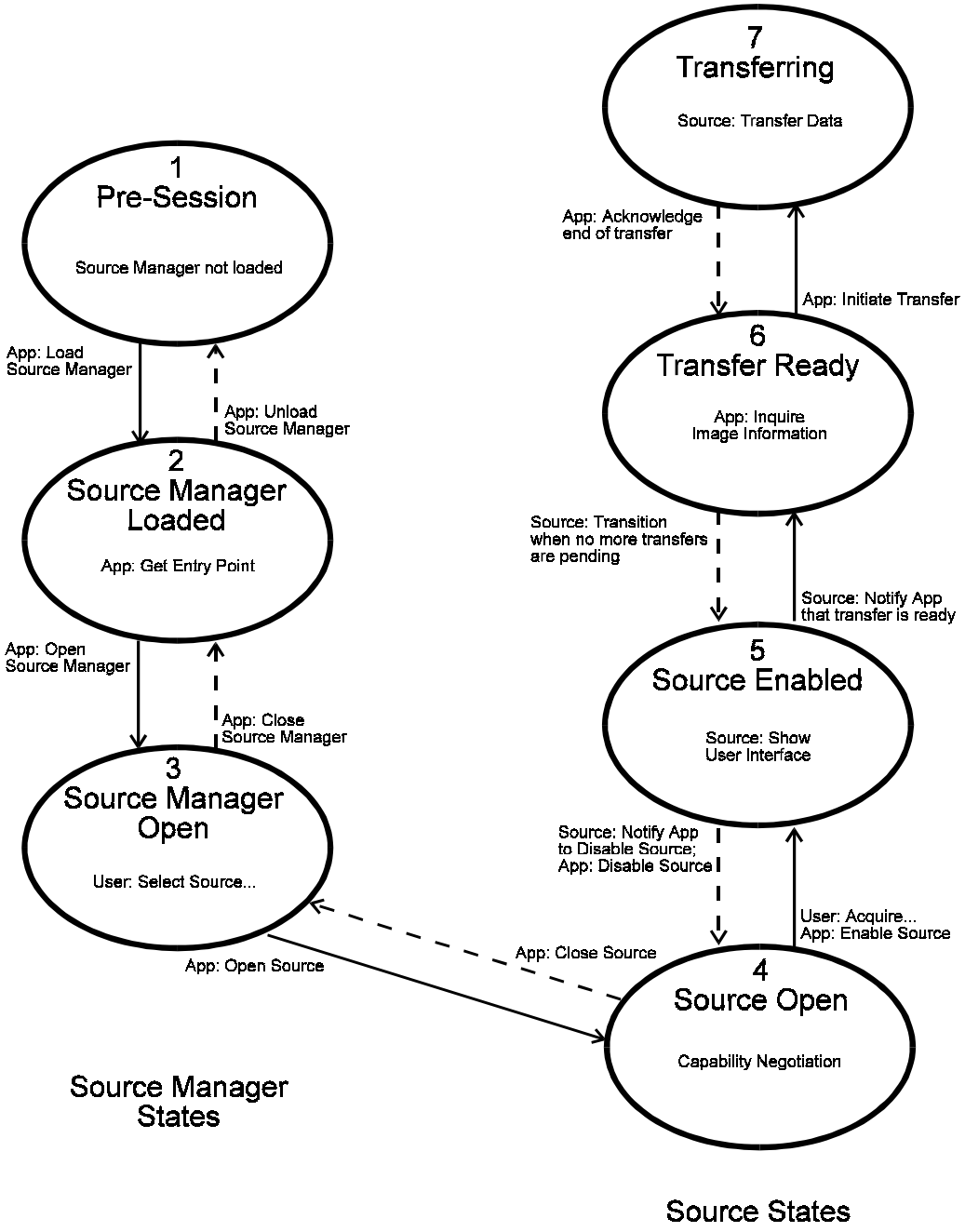


Figure 2-4. State Transition Diagram

The Description of the States

The following sections describe the states.

State 1 - Pre-Session

The Source Manager "resides" in State 1 before the application establishes a session with it.

At this point, the Source Manager code has been installed on the disk but typically is not loaded into memory yet.

The only case where the Source Manager could already be loaded and running is under Windows because the implementation is a DLL (hence, the same instance of the Source Manager can be shared by multiple applications). If that situation exists, the Source Manager will be in State 2 or 3 with the application that loaded it.

State 2 - Source Manager Loaded

The Source Manager now is loaded into memory. It is not open yet.

At this time, the Source Manager is prepared to accept other operation triplets from the application.

State 3 - Source Manager Open

The Source Manager is open and ready to manage Sources.

The Source Manager is now prepared to provide lists of Sources, to open Sources, and to close Sources.

The Source Manager will remain in State 3 for the remainder of the session until it is closed. The Source Manager refuses to be closed while the application has any Sources open.

State 4 - Source Open

The Source has been loaded and opened by the Source Manager in response to an operation from the application. It is ready to receive operations.

The Source should have verified that sufficient resources (i.e. memory, device is available, etc.) exist for it to run.

The application can inquire about the Source's capabilities (i.e. levels of resolution, support of color or black and white images, automatic document feeder available, etc.). The application can also set those capabilities to its desired settings. For example, it may restrict a Source capable of providing color images to transferring black and white only.

Note: Inquiry about a capability can occur while the Source is in States 4, 5, 6, or 7. But, an application can set a capability only in State 4 unless special permission is negotiated between the application and Source.

State 5 - Source Enabled

The Source has been enabled by an operation from the application via the Source Manager and is ready for user-enabled transfers.

If the application has allowed the Source to display its user interface, the Source will do that when it enters State 5.

State 6 - Transfer is Ready

The Source is ready to transfer one or more data items (images) to the application.

The transition from State 5 to 6 is triggered by the Source notifying the application that the transfer is ready.

Before initiating the transfer, the application must inquire information about the image (resolution, image size, etc.).

It is possible for more than one image to be transferred in succession. This topic is covered thoroughly in Chapter 4.

State 7 - Transferring

The Source is transferring the image to the application.

The transfer mechanism being used was negotiated during State 4.

The transfer will either complete successfully or terminate prematurely. The Source sends the appropriate Return Code indicating the outcome.

Once the Source indicates that the transfer is complete, the application must acknowledge the end of the transfer.

Capabilities

One of TWAIN's benefits is it allows applications to easily interact with a variety of acquisition devices. Currently, the devices are limited to image acquisition. However, the variations of capabilities just among image devices is large. For instance,

- Some devices have automatic document feeders.
- Some devices are not limited to one image but can transfer multiple images.
- Some devices support color images.
- Some devices offer a variety of halftone patterns.
- Some devices support a range of resolutions while others may offer different choices.

Developers of applications need to be aware of a Source's capabilities and may influence the capabilities that the Source offers to the application's users. To do this, the application can perform **capability negotiation**. The application generally follows this process:

1. **Determine** if the selected Source supports a particular capability.
2. **Inquire** about the Current Value for this capability. Also, inquire about the capability's Default Value and the set of Available Values that are supported by the Source for that capability.
3. **Request** that the Source set the Current Value to the application's desired value. The Current Value will be displayed as the current selection in the Source's user interface.
4. **Limit**, if needed, the Source's Available Values to a subset of what would normally be offered. For instance, if the application wants only black and white data, it can restrict the Source to transmit only that. If a limitation effects the Source's user interface, the Source should modify the interface to reflect those changes. For example, it may gray out options that are not available because of the application's restrictions.
5. **Verify** that the new values have been accepted by the Source.

TWAIN capabilities are divided into two groups:

- **CAP_xxxx:** Capabilities whose names begin with CAP are capabilities that could apply to any general Source. Such capabilities include use of automatic document feeders, identification of the creator of the data, etc.
- **ICAP_xxxx:** Capabilities whose names begin with ICAP are capabilities that apply to image devices. The I stands for image. (When TWAIN is expanded to support other data transfer such as text or fax data, there will be TCAPs and FCAPs in a similar style.)

Capabilities exist in many varieties but all have a Default Value, Current Value, and may have other values available that can be supported if selected. To help categorize the supported values into clear structures, TWAIN defines four types of **containers** for capabilities.

Name of the Data Structure for the Container	Type of Contents
TW_ONEVALUE	A single value whose current and default values are coincident. The range of available values for this type of capability is simply this single value. For example, a capability that indicates the presence of a document feeder could be of this type.
TW_ARRAY	A rectangular array of values that describe a logical item. It is similar to the TW_ONEVALUE because the current and default values are the same and there are no other values to select from. For example, a list of the names, such as the supported capabilities list returned by the CAP_SUPPORTEDCAPS capability, would use this type of container.
TW_RANGE	Many capabilities allow users to select their current value from a range of regularly spaced values. The capability can specify the minimum and maximum acceptable values and the incremental step size between values. For example, resolution might be supported from 100 to 600 in steps of 50 (100, 150, 200, ..., 550, 600).
TW_ENUMERATION	This is the most general type because it defines a list of values from which the Current Value can be chosen. The values do not progress uniformly through a range and there is not a consistent step size between the values. For example, if a Source's resolution options did not occur in even step sizes then an enumeration would be used (for example, 150, 400, and 600).

In general, most capabilities can have more than one of these containers applied to them depending on how the particular Source implements the capability. The data structure for each of these containers is defined in Chapter 8. A complete table with all defined capabilities is located in Chapter 9. A few of the capabilities must be supported by the application and Source. The remainder of the capabilities are optional.

Modes Available for Data Transfer

There are three different modes that can be used to transfer data from the Source to the application: native, disk file, and buffered memory.

Native

Every Source must support this transfer mode. It is the default mode and is the easiest for an application to implement. However, it is restrictive (i.e. limited to the DIB or PICT formats and limited by available memory).

The format of the data is platform-specific:

- Windows: DIB (Device-Independent Bitmap)
- Macintosh: A handle to a Picture

The Source allocates a single block of memory and writes the image data into the block. It passes a pointer to the application indicating the memory location. The application is responsible for freeing the memory after the transfer.

Disk File

A Source is not required to support this transfer mode but it is recommended.

The application creates the file to be used in the transfer and ensures that it is accessible by the Source for reading and writing.

A capability exists that allows the application to determine which file formats the Source supports. The application can then specify the file format and file name to be used in the transfer.

The disk file mode is ideal when transferring large images that might encounter memory limitations with Native mode. Disk File mode is simpler to implement than the buffered mode discussed next. However, Disk File mode is a bit slower than Buffered Memory mode and the application must be able to manage the file after creation.

Buffered Memory

Every Source must support this transfer mode.

The transfer occurs through memory using one or more buffers. Memory for the buffers are allocated and deallocated by the application.

The data is transferred as an unformatted bitmap. The application must use information available during the transfer (TW_IMAGEINFO and TW_IMAGEMEMXFER) to learn about each individual buffer and be able to correctly interpret the bitmap.

If using the Native or Disk File transfer modes, the transfer is completed in one action. With the Buffered Memory mode, the application may need to loop repeatedly to obtain more than one buffer of data.

Buffered Memory transfer offers the greatest flexibility, both in data capture and control. However, it is the least simple to implement.

3

Application Implementation

This chapter provides the basic information needed to implement TWAIN at a minimum level. In this chapter, you will find information on:

Chapter Contents

Advanced topics are discussed in Chapter 4. They include how to take advantage of Sources that offer automatic feeding of multiple images.

Levels of TWAIN Implementation

Application developers can choose to implement TWAIN features in their application along a range of levels.

- **At the minimum level:** The application does not have to take advantage of capability negotiation or transfer mode selection. Using TWAIN defaults, it can just acquire a single image in the Native mode.
- **At a greater level:** The application can negotiate with the Source for desired capabilities or image characteristics and specify the transfer arrangement. This gives the application more control over the type of image it receives. To do this, developers should follow the instructions provided in this chapter and use information from Chapter 4, as well.
- **At the highest level:** An application may choose to negotiate capabilities, select transfer mode, and create/present its own user interfaces instead of using the built-in ones provided with the Source Manager and Source. Again, refer to this chapter and Chapter 4.

Installation of the Source Manager Software

For a TWAIN-compliant application or Source to work properly, a Source Manager **must** be installed on the host system. To guarantee that a Source Manager is available, ship a copy of the latest Source Manager on your product's distribution disk and provide the user with an installer or installation instructions as suggested below. To ensure that the most recent version of the Source Manager is available to you and your user on their computer, you must do the following:

1. Look for a Source Manager:
   a. **On Windows systems**: Look for the file name TWAIN.DLL in the directory that contains WIN.COM (the Windows directory).
   b. **On Macintosh systems**: Look for the file name "Source Manager" in the TWAIN folder. On System 6, the TWAIN folder lives in the System Folder. On System 7, the TWAIN folder lives in the Preferences folder which lives in the System Folder. (Note, the term "Source Manager" may be localized for other languages.)
2. If no Source Manager is currently installed, install the Source Manager sent out with your application.
3. If a Source Manager already exists, check the version of the installed Source Manager. If the version provided with your application is more recent, rename the existing one as follows and install the Source Manager you shipped. To rename the existing Source Manager:
   a. **On Windows systems**: Rename it to TWAIN.BAK.
   b. **On Macintosh systems**: Rename it to Source Manager Old.

How to Install the Source Manager on MS Windows Systems

To allow the comparison of Source Manager versions, the MS Windows Source Manager DLL has version information built into it which conforms to the Microsoft File Version Stamping specification. Application developers are strongly encouraged to take advantage of this in their installation programs. Microsoft provides the File Version Stamping Library, VER.DLL, which should be used to install the Source Manager.

VER.DLL, VER.LIB and VER.H are included in this Toolkit; VER.DLL may be freely copied and distributed with your installation program. Of course, your installation program will have to link to this DLL to use it. Documentation on the File Version Stamping Library API can be found in the Microsoft Windows 3.1 SDK. VER.DLL can be used under Windows 3.0 as well as 3.1.

The following code fragment demonstrates how the VerInstallFile( ) function provided in VER.DLL can be used to install the Source Manager into the user's Windows directory.

Note that the following example assumes that your installation floppy disk is in the A: drive and the Source Manager is in the root of the installation disk.

```
#include "windows.h"
#include "ver.h"
#include "stdio.h"

// Max file name length is based on 8 dot 3 file name convention.
#define MAXFNAMELEN 12
// Max path name length is based on GetWindowsDirectory()
// documentation.
#define MAXPATHLEN 144

VOID InstallWinSM ( VOID )
{
   DWORD  dwInstallResult;
   WORD   wTmpFileLen = MAXPATHLEN;
   WORD   wLen;

   char   szSrcDir[MAXPATHLEN];
   char   szDstDir[MAXPATHLEN];
   char   szCurDir[MAXPATHLEN];
   char   szTmpFile[MAXPATHLEN];

   wLen = GetWindowsDirectory( szDstDir, MAXPATHLEN );
   if (!wLen || wLen>MAXPATHLEN)
   {
      return;   // failure getting Windows dir
   }

   strcpy( szCurDir, szDstDir );
   strcpy( szSrcDir, "a:\\" );

   dwInstallResult = VerInstallFile( VIFF_DONTDELETEOLD,
                                     "TWAIN.DLL",
                                     "TWAIN.DLL",
                                     szSrcDir,
                                     szDstDir,
                                     szCurDir,
                                     szTmpFile,
                                     &wTmpFileLen );

   // If VerInstallFile() left a temporary copy of the new
   // file in DstDir be sure to delete it. This happens
   // when a more recent version is already installed.
   if ( dwInstallResult & VIF_TEMPFILE  &&
       ((wTmpFileLen - MAXPATHLEN) > MAXFNAMELEN) )
   {
      // when dst path is root it already ends in '\'
      if (szDstDir[wLen-1] != '\\')
      {
         strcat( szDstDir, "\\" );
      }
      strcat( szDstDir, szTmpFile );
      remove( szDstDir );
   }
}
```

You probably wish to enhance the above code so it handles low memory and other error conditions, as indicated by the dwInstallResult return code. Also note that the above code does not leave a backup copy of the user's prior Source Manager on their disk but you should do this. Copy the older version to TWAIN.BAK.

How to Install the Source Manager on Macintosh Systems

To perform the comparison of Source Manager versions, use the vers resource.

Use the following algorithm to develop code for installing the Source Manager on the Macintosh:

```
if (there is not a Source Manager installed) or
   (if installed Source Manager is older)
   if (the current system predates 7.0)
      if (there isn't a "TWAIN" folder in the System Folder)
         Create a "TWAIN" folder in the System Folder
      copy the Source Manager into the "TWAIN" folder in System Folder
   else
      if (there isn't a "TWAIN" folder in the Preferences Folder)
         Create a "TWAIN" folder in the Preferences folder
      copy the Source Manager into the "TWAIN" folder
```

Changes Needed to Prepare for a TWAIN Session

There are three areas of the application that must be changed before a TWAIN session can even begin. The application developer must:

1. Alter the application's user interface to add Select Source and Acquire menu choices
2. Include the file called TWAIN.H in your application
3. Alter the application's event loop

Alter the Application's User Interface to Add Select Source and Acquire Options

As mentioned in the Technical Overview chapter, the application should include two menu items in its File menu: **Select Source...** and **Acquire...**. It is strongly recommended that you use these phrases since this consistency will benefit all users.

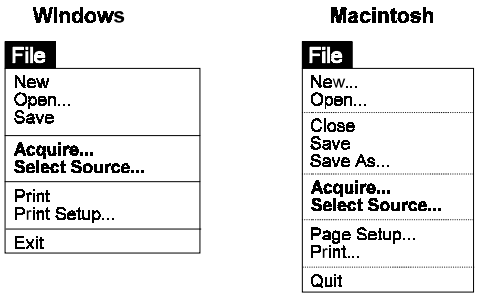


Figure 3-1. User Interface for Selecting a Source and Acquiring Options

Note the following:

When this is selected:	The application does this:
Select Source...	The application requests that the Source Manager's Select Source Dialog Box appear (or it may display its own version). After the user selects the Source they want to use, control returns to the application.
Acquire...	The application requests that the Source display its user interface. (Again, the application can create its own version of a user interface or display no user interface.)

Detailed information on the operations used by the application to successfully acquire data is provided later in this chapter in the section called "Controlling a TWAIN Session from your Application."

Include the TWAIN.H File in Your Application

The TWAIN.H file that is shipped with this TWAIN Developer's Toolkit contains all of the critical definitions needed for writing a TWAIN-compliant application or Source. Be sure to include it in your application's code and print out a copy to refer to while reading this chapter.

The TWAIN.H file contains:

Category	Prefix for each item
Data Groups	DG_
Data Argument Types	DAT_
Messages	MSG_
Capabilities	CAP_ or ICAP_
Return Codes	TWRC_
Condition Codes	TWCC_
Type Definitions	TW_
Structure Definitions	TW_
Entry points	These are DSM_Entry and DS_Entry

In addition, there are many constants defined in TWAIN.H which are not listed here.

Alter the Application's Event Loop

Events include activities such as key clicks, mouse events, periodic events, accelerators, etc. Every TWAIN-compliant application, whether on Macintosh or Windows, needs an event loop. (On Windows, these actions are called messages but that can be confusing because TWAIN uses the term messages to describe the third parameter of an operation triplet. Therefore, we will refer to these key clicks, etc. as events in this section generically for both Windows and Macintosh.)

During a TWAIN session, the application opens one or more Sources. However, even if several Sources are open, the application should only have one Source enabled at any given time. That is the Source from which the user is attempting to acquire data.

Altering the event loop serves two purposes:

- Passing events from the application to the Source so it can respond to them
- Notifying the application when the Source is ready to transfer data or have its user interface disabled

Event Loop Modification - Passing events (The first purpose)

While a Source is enabled, all events are sent to the application's event loop. Some of the events may belong to the application but others belong to the enabled Source. To ensure that the Source receives and processes its events, the following changes are required:

The application **must** send all events that it receives in its event loop to the Source as long as the Source is enabled. The application uses:

DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT

The TW_EVENT data structure used looks like this:

```
typedef struct {
   TW_MEMREF     pEvent;     /* Windows pMSG or MAC pEvent   */
   TW_UINT16     TWMessage;  /* TW message from Source to    */
                             /* the application              */
} TW_EVENT, FAR *pTW_EVENT;
```

The pEvent field points to the EventRecord (Macintosh) or message structure (Windows).

The Source receives the event from the Source Manager and determines if the event belongs to it.

- **If it does**, the Source processes the event. It then sets the Return Code to TWRC_DSEVENT to indicate it was a Source event. In addition, it should set the TWMessage field of the TW_EVENT structure to MSG_NULL.
- **If it does not**, the Source sets the Return Code to TWRC_NOTDSEVENT meaning it is not a Source event. In addition, it should set the TWMessage field of the TW_EVENT structure to MSG_NULL. The application receives this information from DSM_Entry and should process the event in its event loop as normal.

On Macintosh only, the application must periodically send NULL events to the Source to allow notifications from Source to application.

Event Loop Modification - Notifications from Source to application (The second purpose)

When the Source has data ready for a data transfer or it wishes to request that its user interface be disabled, it needs to communicate this information to the application asynchronously.

These notifications appear in the application's event loop. They are contained in the TW_EVENT.TWMessage field. The two notices of interest are:

MSG_XFERREADY to indicate data is ready for transfer

MSG_CLOSEDSREQ to request that the Source's user interface be disabled

Therefore, the application's event loop must always check the TW_EVENT.TWMessage field following a DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT call to determine if it is the simple MSG_NULL or critical MSG_XFERREADY or MSG_CLOSEDSREQ. Information about how the application should respond to these two special notices is detailed later in this chapter in the "Controlling a TWAIN Session from your Application" section.

How to Modify the Event Loop for MS Windows

This section illustrates typical modifications needed in an MS Windows application to support TWAIN-connected Sources.

```
TW_EVENT   twEvent;
TW_INT16   rc;

while (GetMessage ( (LPMSG) &msg, NULL, 0, 0) )
{
      if Source is enabled
      {
            twEvent.pEvent = (TW_MEMREF)&msg;
            twEvent.TWMessage = MSG_NULL;
            rc = (*pDSM_Entry) (pAppId,
                                pSourceId,
                                DG_CONTROL,
                                DAT_EVENT,
                                MSG_PROCESSEVENT,
                               (TW_MEMREF)&twEvent);

            // check for message from Source
            switch (twEvent.TWMessage)
            {
                  case MSG_XFERREADY:
                        SetupAndTransferImage(NULL);
                        break;

                  case MSG_CLOSEDSREQ:
                        DisableAndCloseSource(NULL);
                        break;

                  case MSG_NULL:
                        // no message was returned from the source
                        break;
            }

            if (rc == TWRC_NOTDSEVENT)
            {
                  TranslateMessage( (LPMSG) &msg);
                  DispatchMessage( (LPMSG) &msg);
            }

      }
}
```

Note: Source writers are advised to keep stack space usage to a minimum. Application writers should be also be aware that, in the Windows environment, sources run in their calling application's data space. They depend upon the application to reserve enough stack space for the source to be able to perform its various functions. For this reason, applications should define enough stack space in their linker DEF files for the sources that they might use.

How to Modify the Event Loop for Macintosh

This section illustrates typical modifications needed in a Macintosh application to support TWAIN-connected Sources.

```
TW_EVENT      twEvent;
TW_INT16      rc;
EventRecord   theEvent;
while (!Done){
      If Source is Enabled{
            //Send periodic NULL events to the Source
            twEvent.pEvent = NULL;
            twEvent.TWMessage = MSG_NULL;
            rc = (*pDSM_Entry) (pAppID,
                        pSourceID,
                        DG_CONTROL,
                        DAT_EVENT,
                        MSG_PROCESSEVENT,
                        (TW_MEMREF)&twEvent);
            //check for message from Source
            switch (twEvent.TWMessage){
                  case MSG_XFERREADY:
                        SetupImage(NULL);
                        break;
                  case MSG_CLOSEDSREQ:
                        DisableSource(NULL);
                        break;
                  case MSG_NULL:
                        //no message was returned from the Source
                        break;
            }
      }
      if (GetNextEvent(everyEvent, &theEvent) ){   //or WaitNextEvent()
            If Source is Enabled{
                  twEvent.pEvent = &theEvent;
                  twEvent.TWMessage = MSG_NULL;
                  rc = (*pDSM_Entry) (pAppID,
                        pSourceID,
                        DG_CONTROL,
                        DAT_EVENT,
                        MSG_PROCESSEVENT,
                       (TW_MEMREF)&twEvent);
                  //check for message from Source
                  switch (twEvent.TWMessage){
                        case MSG_XFERREADY:
                              SetupImage(NULL);
                              break;
                        case MSG_CLOSEDSREQ:
                              DisableSource(NULL);
                              break;
                        case MSG_NULL:
                              //no message was returned from the Source
                              break;
                  }
```

```
                    if (rc == TWRC_NOTDSEVENT)
                          Message=DealWithEvent(&theEvent);
              }
        } else
              Message=DealWithEvent(&theEvent);
}
```

The DSM_Entry Call and Available Operation Triplets

As described in the Technical Overview chapter, all actions that the application invokes on the Source Manager or Source are routed through the Source Manager. The application passes the request for the action to the Source Manager via the DSM_Entry function call which contains an operation triplet describing the requested action.

In code form, the DSM_Entry function looks like this:

On Windows:

```
TW_UINT16 FAR PASCAL DSM_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       pTW_IDENTITY   pDest,      // destination of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

On Macintosh:

```
FAR PASCAL TW_UINT16 DSM_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       pTW_IDENTITY   pDest,      // destination of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

The DG, DAT, and MSG parameters contain the operation triplet.

The parameters must follow these rules:

pOrigin

References the application's TW_IDENTITY structure. The contents of this structure must not be changed by the application from the time the connection is made with the Source Manager until it is closed.

pDest

Set to NULL if the operation's final destination is the Source Manager.

Otherwise, set to point to a valid TW_IDENTITY structure for an open Source.

DG_xxxx

Data Group of the operation. Currently, only DG_CONTROL and DG_IMAGE are defined. Custom Data Groups can be defined.

DAT_xxxx

Designator that uniquely identifies the type of data "object" (structure or variable) referenced by pData.

MSG_xxxx

Message specifies the action to be taken.

pData

Refers to the TW_xxxx structure or variable that will be used during the operation. Its type is specified by the DAT_xxxx. This parameter should always be typecast to TW_MEMREF when it is being referenced.

Operation Triplets - Application to Source Manager

There are nine operation triplets that can be sent from the application to be consumed by the Source Manager. They all use the DG_CONTROL data group and they use three different data argument types: DAT_IDENTITY, DAT_PARENT, and DAT_STATUS. The following table lists the data group, data argument type, and messages that make up each operation. The list is in alphabetical order not the order in which they are typically called by an application. Details about each operation are available in reference format in Chapter 7.

Control Operations from Application to Source Manager

DG_CONTROL / DAT_IDENTITY

MSG_CLOSEDS :	Prepare specified Source for unloading
MSG_GETDEFAULT :	Get identity information of the default Source
MSG_GETFIRST :	Get identity information of the first available Source
MSG_GETNEXT :	Get identity of the next available Source
MSG_OPENDS :	Load and initialize the specified Source
MSG_USERSELECT :	Present "Select Source" dialog

DG_CONTROL/DAT_PARENT

MSG_CLOSEDSM :	Prepare Source Manager for unloading
MSG_OPENDSM :	Initialize the Source Manager

DG_CONTROL/DAT_STATUS

MSG_GET :	Return Source Manager's current Condition Code

Operation Triplets - Application to Source

The next group of operations are sent to a specific Source by the application. These operations are still passed via the Source Manager using the DSM_Entry call. The first set of triplets use the DG_CONTROL identification for their data group. These are operations that could be performed on any kind of TWAIN device. The second set of triplets use the DG_IMAGE identification for their data group which indicates these operations are specific to image data. Details about each operation are available in reference format in Chapter 7.

Control Operations from Application to Source

DG_CONTROL / DAT_CAPABILITY

MSG_GET :	Return a Capability's valid value(s) including current and default values
MSG_GETCURRENT :	Get a Capability's current value
MSG_GETDEFAULT :	Get a Capability's preferred default value (Source specific)
MSG_RESET :	Change a Capability's current value to its TWAIN-defined default (See Chapter 9)
MSG_SET :	Change a Capability's current and/or available value(s)

DG_CONTROL/ DAT_EVENT

MSG_PROCESSEVENT :	Pass an event to the Source from the application

DG_CONTROL/ DAT_PENDINGXFERS

MSG_ENDXFER :	Application acknowledges or requests the end of data transfer
MSG_GET :	Return the number of transfers the Source is ready to supply
MSG_RESET :	Set the number of pending transfers to zero

DG_CONTROL/ DAT_SETUPFILEXFER

MSG_GET :	Return info about the file that the Source will write the acquired data into
MSG_GETDEFAULT :	Return the default file transfer information
MSG_RESET :	Reset current file information to default values
MSG_SET :	Set file transfer information for next file transfer

DG_CONTROL/ DAT_SETUPMEMXFER

MSG_GET :	Return Source's preferred, minimum, and maximum transfer buffer sizes

DG_CONTROL/ DAT_STATUS

MSG_GET :	Return the current Condition Code from specified Source

DG_CONTROL/ DAT_USERINTERFACE

MSG_DISABLEDS :	Cause Source's user interface to be taken down
MSG_ENABLEDS :	Cause Source to prepare to display its user interface

DG_CONTROL/ DAT_XFERGROUP

MSG_GET :	Return the Data Group (currently DG_IMAGE or a custom data group) for the upcoming transfer

There are five more DG_CONTROL operations for communications between the Source Manager and the Source. They are discussed in Chapter 5.

Image Operations from Application to Source

DG_IMAGE / DAT_CIECOLOR

MSG_GET :	Return the CIE XYZ information for the current transfer

DG_IMAGE / DAT_GRAYRESPONSE

MSG_RESET :	Reinstate identity response curve for grayscale data
MSG_SET : Source uses specified response curve on grayscale data	

DG_IMAGE / DAT_IMAGEFILEXFER

MSG_GET :	Initiate image acquisition using the Disk File transfer mode

DG_IMAGE / DAT_IMAGEINFO

MSG_GET :	Return information that describes the image for the next transfer

DG_IMAGE / DAT_IMAGELAYOUT

MSG_GET :	Describe physical layout / position of "original" image
MSG_GETDEFAULT :	Default information on the layout of the image
MSG_RESET :	Set layout information for the next transfer to defaults
MSG_SET :	Set layout for the next image transfer

DG_IMAGE / DAT_IMAGEMEMXFER

MSG_GET :	Initiate image acquisition using the Buffered Memory transfer mode

DG_IMAGE / DAT_IMAGENATIVEXFER

MSG_GET :	Initiate image acquisition using the Native transfer mode

DG_IMAGE / DAT_JPEGCOMPRESSION

MSG_GET :	Return JPEG compression parameters for current transfer
MSG_GETDEFAULT :	Return default JPEG compression parameters
MSG_RESET :	Use Source's default JPEG parameters on JPEG transfers
MSG_SET :	Use specified JPEG parameters on JPEG transfers

DG_IMAGE / DAT_PALETTE8

MSG_GET :	Return palette information for current transfer
MSG_GETDEFAULT :	Return Source's default palette information for current pixel type
MSG_RESET :	Use Source's default palette for transfer of this pixel type
MSG_SET :	Use specified palette for transfers of this pixel type

DG_IMAGE / DAT_RGBRESPONSE

MSG_RESET :	Use Source's default (identity) RGB response curve
MSG_SET :	Use specified response curve for RGB transfers

DSM_Entry Parameters

The parameters for the DG_xxxx, DAT_xxxx, and MSG_xxxx fields are determined by the operation triplet. The other parameters are filled as follows:

pOrigin

Refers to a copy of the application's TW_IDENTITY structure.

pDest

If the operation's destination is the Source Manager: Always holds a value of NULL. This indicates to the Source Manager that the operation is to be consumed by it not passed on to a Source.

If the operation's destination is a Source: This parameter references a copy of the Source's TW_IDENTITY structure that is maintained by the application. The application received this structure in response to the DG_CONTROL / DAT_IDENTITY / MSG_OPENDS operation sent from the application to the Source Manager. This is discussed more in the next section (Controlling a TWAIN Session from your Application - State 3 to 4).

pData

Always references a structure or variable corresponding to the TWAIN type specified by the DAT_xxxx parameter. Typically, but not always, the data argument type name corresponds to a TW_xxxx data structure name. For example, the DAT_IDENTITY argument type uses the corresponding TW_IDENTITY data structure. All data structures can be seen in the file called TWAIN.H. The application is responsible for allocating and deallocating the structure or element and assuring that pData correctly references it.

Note that there are two cases when the Source, rather than the application, allocates a structure that is used during an operation.

- One occurs during DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, and MSG_RESET operations. The application still allocates **pData* but the Source allocates a structure referenced by **pData* called a "container structure".
- The other occurs during the DG_IMAGE / DAT_JPEGCOMPRESSION operations. The topic of data compression is covered in Chapter 4.

In all cases, the application still deallocates all structures.

Controlling a TWAIN Session from Your Application

In addition to the preparations discussed at the beginning of this chapter, the application must be modified to actually initiate and control a TWAIN session.

The session consists of the seven states of the TWAIN protocol as introduced in the Technical Overview. However, the application is not forced to move the session from State 1 to State 7 without stopping. For example, some applications may choose to pause in State 3 and move among the higher states (4 - 7) to repeatedly open and close Sources when acquisitions are requested by the user. Another example of session flexibility occurs when an application transfers multiple images during a session. The application will repeatedly move the session from State 6 to State 7 then back to State 6 and forward to State 7 again to transfer the next image.

For the sake of simplicity, this chapter illustrates moving the session from State 1 to State 7 and then backing it out all the way from State 7 to State 1. The diagram on the next page shows the operation triplets that are used to transition the session from one state to the next. Detailed information about each state and its associated transitions follow. The topics include:

- Load the Source Manager and Get the DSM_Entry (State 1 to 2)
- Open the Source Manager (State 2 to 3)
- Select the Source (during State 3)
- Open the Source (State 3 to 4)
- Negotiate Capabilities with the Source (during State 4)
- Request the Acquisition of Data from the Source (State 4 to 5)
- Recognize that the Data Transfer is Ready (State 5 to 6)
- Start and Perform the Transfer (State 6 to 7)
- Conclude the Transfer (State 7 to 6 to 5)
- Disconnect the TWAIN Session (State 5 to 1 in sequence)

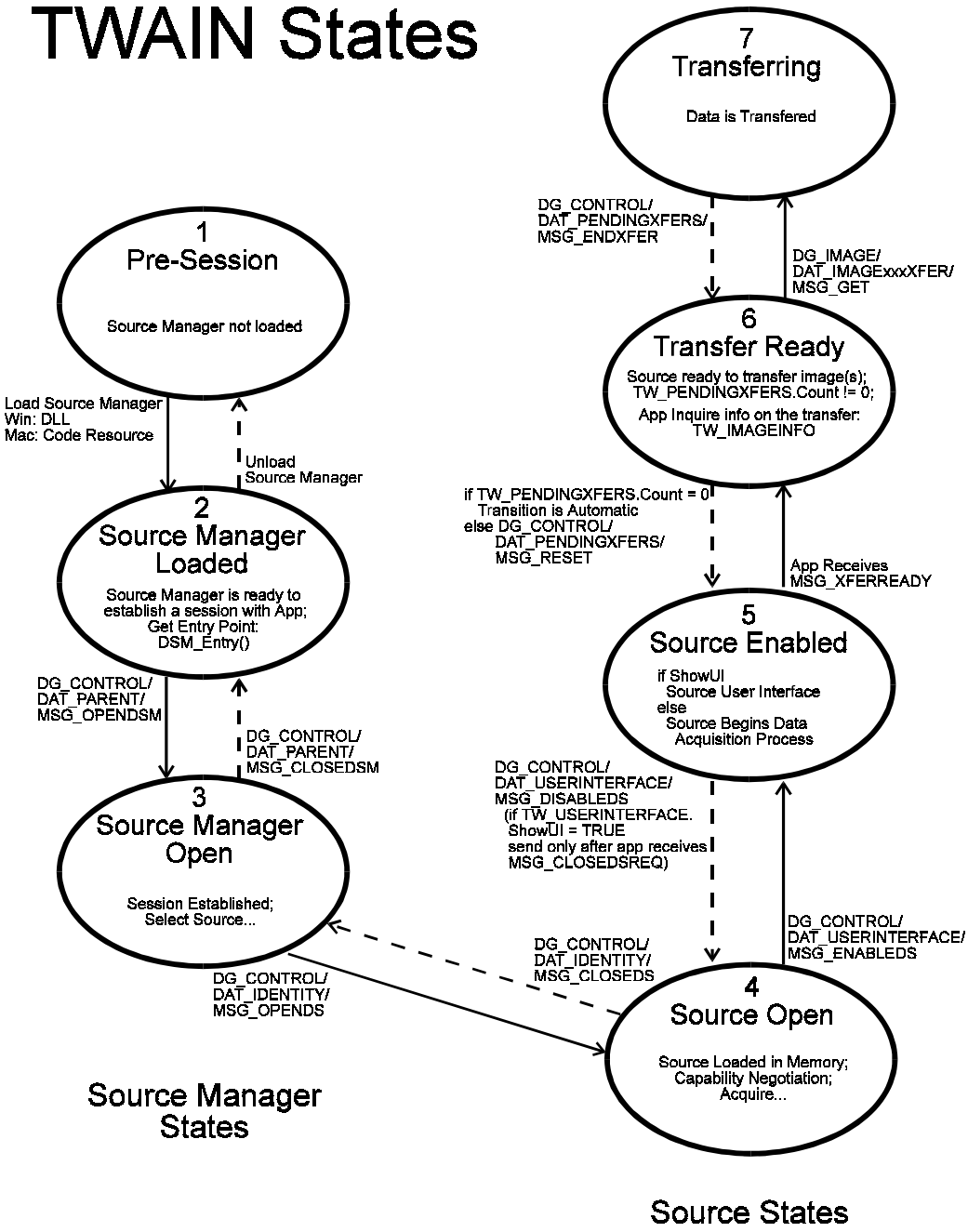


Figure 3-2. TWAIN States

State 1 to 2 - Load the Source Manager and Get the DSM_Entry

The application must load the Source Manager before it is able to call its DSM_Entry point.

Operations Used:

No TWAIN operations are used for this transition. Instead,

On Windows:

Load TWAIN.DLL using the LoadLibrary( ) routine.

Get the DSM_Entry by using the GetProcAddress( ) call.

On Macintosh:

Open the Source Manager code resource using OpenResFile( ) and GetResource( ).

Get the DSM_Entry by dereferencing the handle to the Source Manager.

On Windows:

The application can load the Source Manager by doing the following:

```
DSMENTRYPROC   pDSM_Entry;
HANDLE         hDSMLib;
char           szSMDir;
OFSTRUCT       of;

// check for the existence of the TWAIN.DLL file in the Windows directory
   GetWindowsDirectory (szSMDir, sizeof(szSMDir));
   /*** Could have been networked drive with trailing '\' ***/
   if (szSMDir [(lstrlen (szSMDir) - 1)] != '\\')
   {      lstrcat( szSMDir, "\\" );
   }
if ((OpenFile(szSMDir, &of, OF_EXIST) != -1)
{
   // load the DLL
   if (hDSMDLL = LoadLibrary(DSMName)) != NULL)
   {
      // check if library was loaded
      if (hDSMDLL >= (HANDLE)VALID_HANDLE)
      {
         if (lpDSM_Entry = (DSMENTRYPROC)GetProcAddress(hDSMDLL,
MAKEINTRESOURCE (1))) != NULL)
         {
            if (lpDSM_Entry )
               FreeLibrary(hDSMDLL);
         }
      }
   }
}
```

Note, the code appends TWAIN.DLL to the end of the Windows directory and verifies that the file exists before calling LoadLibrary( ). Applications are strongly urged to perform a dynamic run-time link to DSM_Entry( ) by calling LoadLibrary( ) rather than statically linking to TWAIN.LIB via the linker. If the TWAIN.DLL is not installed on the machine, MS Windows will fail to load an application that statically links to TWAIN.LIB. If the Application has a dynamic link, however, it will be able to give users a meaningful error message, and perhaps continue with image acquisition facilities disabled.

After getting the DSM_Entry, the application must check pDSM_Entry. If it is NULL, it means that the Source Manager has not been installed on the user's machine and the application cannot provide any TWAIN services to the user. If NULL, the application must not attempt to call *pDSM_Entry as this would result in an Unrecoverable Application Error (UAE).

On Macintosh:

The Source Manager, named Source Manager, is a code resource on the Macintosh (its resource type is DSMR). Under System 6, this file resides within the TWAIN folder which lives in the System Folder. Under System 7, the TWAIN folder lives in the Preferences folder which lives in the System Folder. To access the Source Manager, the application must first load and lock down this code resource. The application must call the Source Manager (via the DSM_Entry() call) as a Pascal function.

The following code segments were written in Think "C" v5.0. Your exact coding may vary but the approach will be consistent.

The following code segment will load the Source Manager:

```
typedef PASCAL TW_UINT16(*DSMEntryFunc)(pTW_IDENTITY,
                                        pTW_IDENTITY,
                                        TW_UINT32,
                                        TW_UINT16,
                                        TW_UINT16,
                                        TW_MEMREF);

DSMEntryFunc pDSM_Entry;
TW_INT16   SaveRes;

SaveRes=CurResFile();          /* save the current resource file  */

/* Be sure to change the working directory to the TWAIN folder    */
DSMRefNum=OpenResFile(DSMName); /* open the Source Manager file   */

DSMHandle=GetResource(DSMR_type,TWON_DSMCODEID); /* get the SM resource*/
HLock(DSMHandle);               /* lock the SM resource handle     */
pDSM_Entry=(pTW_INT16)*DSMHandle; /* get pointer to the DSM_Entry  */

/* Be sure to restore the working directory */
UseResFile(SaveRes);           /* restore current resource file */
```

After getting the DSM_Entry, application must check pDSM_Entry. If it is NULL, it means that the Source Manager has not been installed on the user's machine and the application cannot provide any TWAIN services to the user. If NULL, do not attempt to call it.

State 2 to 3 - Open the Source Manager

The Source Manager has been loaded. The application must now open the Source Manager.

One Operation is Used:

DG_CONTROL / DAT_PARENT / MSG_OPENDSM

pOrigin

The application must allocate a structure of type TW_IDENTITY and fill in all fields except for the Id field. Once the structure is prepared, this pOrigin parameter should point at that structure.

During the MSG_OPENDSM operation, the Source Manager will fill in the Id field with a unique identifier of the application. The value of this identifier is only valid while the application is connected to the Source Manager.

The application must save the entire structure. From now on, the structure will be referred to by the pOrigin parameter to identify the application in every call the application makes to DSM_Entry( ).

The TW_IDENTITY structure is defined in the TWAIN.H file but for quick reference, it looks like this:

```
/* DAT_IDENTITY Identifies the program/library/code */
/*              resource.                           */
typedef struct {
   TW_UINT32    Id; /* Unique number for identification*/
   TW_VERSION   Version;
   TW_UINT16    ProtocolMajor;
   TW_UINT16    ProtocolMinor;
   TW_UINT32    SupportedGroups;/*Bit field OR combination */
                                /*of DG_constants found in */
                                /*the TWAIN.H file         */
   TW_STR32     Manufacturer;
   TW_STR32     ProductFamily;
   TW_STR32     ProductName;
} TW_IDENTITY, FAR *pTW_IDENTITY;
```

pDest

Set to NULL indicating the operation is intended for the Source Manager.

pData

Typically, you would expect to see this point to a structure of type TW_PARENT but this is not the case. This is an exception to the usual situation where the DAT field of the triplet identifies the data structure for pData.

- **On Windows:** pData points to the window handle (hWnd) that will act as the Source's "parent". The variable is of type TW_INT32. For 16 bit MS Windows, the handle is stored in the low word of the 32 bit integer and the upper word is set to zero. If running under the WIN32 environment, this is a 32 bit window handle. The Source Manager will maintain a copy of this window handle for posting messages back to the application.
- **On Macintosh:** pData should be a 32-bit NULL value.

How to Initialize the TW_IDENTITY Structure

Here is a Windows example of code used to initialize the application's TW_IDENTITY structure.

```
TW_IDENTITY     AppID;       // Application identity structure
   AppID.Id = 0; //Initialize to 0 (Source Manager will assign real
                 // value)
   AppID.Version.MajorNum = 3;   //Your application's version number
   AppID.Version.MinorNum = 5;
   AppID.Version.Language = TWLG_USA;
   AppID.Version.Country  = TWCY_USA;
   lstrcpy (AppID.Version.Info, "Your Application's Version String");
   AppID.ProtocolMajor = TWON_PROTOCOLMAJOR;
   AppID.ProtocolMinor = TWON_PROTOCOLMINOR;
   AppID.SupportedGroups = DG_IMAGE | DG_CONTROL;
   lstrcpy (AppID.Manufacturer, "Application's Manufacturer");
   lstrcpy (AppID.ProductFamily, "Application's Product Family");
   lstrcpy (AppID.ProductName, "Specific Application Product Name");
```

On Windows: Using DSM_Entry to open the Source Manager

```
TW_UINT16  rc;
rc = (*pDSM_Entry) (&AppID,
                    NULL,
                    DG_CONTROL,
                    DAT_PARENT,
                    MSG_OPENDSM,
                    (TW_MEMREF) &hWnd);
```

where AppID is the TW_IDENTITY structure that the application set up to identify itself and hWnd is the application's main window handle.

On Macintosh: Using DSM_Entry to open the Source Manager

```
TW_UINT16  SaveRes;

SaveRes=CurResFile();  /* Save the current resource file */
UseResFile(DSMRefNum); /* Set Source Manager resource file as
                       /* current*/

rc = (*pDSM_Entry) (&AppID,
                    NULL,
                    DG_CONTROL,
                    DAT_PARENT,
                    MSG_OPENDSM,
                    NULL);

UseResFile(SaveRes);   /* Restore the resource file */
```

where AppID is the TW_IDENTITY structure that the application set up to identify itself. The same approach is used by the application to call the DSM_Entry( ) function throughout the remainder of the TWAIN session.

Note: **Check the Return Code:** Whenever your application uses a TWAIN operation, **always check the Return Code** sent back through the DSM_Entry function to be certain an operation is successful. If an operation indicates failure, use the DG_CONTROL / DAT_STATUS / MSG_GET operation to get the Condition Code that indicates the cause of failure. The application identifies who the status triplet be sent to, the Source Manager or Source, depending on which one received the original operation that failed.

State 3 - Select the Source

The Source Manager has just been opened and is now available to assist your application in the selection of the desired Source.

One Operation is Used:

DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT

pOrigin

Points to the application's TW_IDENTITY structure. The desired data type should be specified by the application. This was done when you initialized the SupportedGroups field in your application's TW_IDENTITY structure.

This causes the Source Manager to make available for selection by the user only those Sources that can provide the requested data type(s). All other Sources are grayed out. (Note, if more than one data type were available, for example image and text, and the application wanted to accept both types of data, it would do a bit-wise OR of the types' constants and place the results into the SupportedGroups field.)

pDest

Set to NULL.

pData

Points to a structure of type TW_IDENTITY. The application must allocate this structure prior to making the call to DSM_Entry. Once the structure is allocated, the application must:

- Set the Id field to zero.
- Set the ProductName field to the null string ("\0"). (If the application wants a specific Source to be highlighted in the Select Source dialog box, other than the system default, it can enter the ProductName of that Source into the ProductName field instead of null. The system default Source and other available Sources can be determined by using the DG_CONTROL / DAT_IDENTITY / MSG_GETDEFAULT, MSG_GETFIRST and MSG_GETNEXT operations.)

Additional fields of the structure will be filled in by the Source Manager during this operation to identify the selected Source. Make sure the application keeps a copy of this updated structure after completing this call. You will use it to identify the Source from now on.

The most common approach for selecting the Source is to use the Source Manager's Select Source dialog box. This is typically displayed when the user clicks on your Select Source option. To do this:

1. The application sends a DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT operation to the Source Manager to have it display its dialog box. The dialog displays a list of all Sources that are installed on the system that can provide data of the type specified by the application. It highlights the Source that is the system default unless the application requests otherwise.
2. The user selects a Source or presses the Cancel button. If no devices are available, the Select Source Dialog's Select/OK button will be grayed out and the user will have no choice but to select Cancel.
3. The application must check the Return Code of DSM_Entry to determine the user's action.
    a. **If TWRC_SUCCESS**: Their selected Source is listed in the TW_IDENTITY structure pointed to by the pData parameter and is now the default Source.
    b. **If TWRC_CANCEL**: The user either clicked Cancel intentionally or had no other choice because no devices were listed. Do not attempt to open a Source.
    c. **If TWRC_FAILURE**: Use the DG_CONTROL / DAT_STATUS / MSG_GET operation (sent to the Source Manager) to determine the cause. The most likely cause is a lack of sufficient memory.

As an alternative to using the Source Manager's Select Source dialog, the application can devise its own method for selecting a Source. For example, it could create and display its own user interface or simply select a Source without offering the user a choice. This alternative is discussed in Chapter 4.

State 3 to 4 - Open the Source

The Source Manager is open and able to help your application open a Source.

One Operation is Used:

DG_CONTROL / DAT_IDENTITY / MSG_OPENDS

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Set to NULL.

pData

Points to a structure of type TW_IDENTITY.

Typically, this points to the application's copy of the Source's TW_IDENTITY structure filled in during the MSG_USERSELECT operation previously.

However, if the application wishes to have the Source Manager simply open the default Source, it can do this by setting the TW_IDENTITY.ProductName field to "\0" (null string) and the TW_IDENTITY.Id field to zero.

During the MSG_OPENDS operation, the Source Manager assigns a unique identifier to the Source and records it in the TW_IDENTITY.Id field. Copy the resulting TW_IDENTITY structure. Once the Source is opened, the application will point to this resulting structure via the pDest parameter on every call that the application makes to DSM_Entry where the desired destination is this Source.

Note: The user is not required to take advantage of the Select Source option. They may click on the Acquire option without having selected a Source. In that case, your application should open the default Source. **The default source is either the last one used by the user or the last one installed.**

State 4 - Negotiate Capabilities with the Source

At this point, the application has a structure identifying the open Source. Operations can now be directed from the application to that Source. To receive a single image from the Source, only one capability, CAP_XFERCOUNT, must be negotiated now. All other capability negotiation is optional.

Two Operations are Used:

DG_CONTROL / DAT_CAPABILITY / MSG_GET

DG_CONTROL / DAT_CAPABILITY / MSG_SET

The parameters for each of the operations, in addition to the triplet, are these:

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Points to the desired Source's TW_IDENTITY structure. The Source Manager will receive the DSM_Entry call, recognize that the destination is a Source rather than itself, and pass the operation along to the Source via the DS_Entry function.

pData

Points to a structure of type TW_CAPABILITY.

The definition of TW_CAPABILITY is:

```
typedef struct {
  TW_UINT16   Cap;       /* ID of capability to get or set */
  TW_UINT16   ConType;   /* TWON_ONEVALUE, TWON_RANGE,     */
                         /* TWON_ENUMERATION or TWON_ARRAY */
  TW_HANDLE  hContainer; /* Handle to container of type    */
                         /* ConType                        */
} TW_CAPABILITY, FAR *pTW_CAPABILITY;
```

The Source allocates the container structure pointed to by the hContainer field when called by the MSG_GET operation. The application allocates it when calling with the MSG_SET operation. Regardless of who allocated it, the application deallocates the structure either when the operation is complete or when the application no longer needs to maintain the information.

Each operation serves a special purpose:

MSG_GET

Since Sources are not required to support all capabilities, this operation can be used to determine if a particular TWAIN-defined capability is supported by a Source. The application needs to set the Cap field of the TW_CAPABILITY structure to the identifier representing the capability of interest. The constants identifying each capability are listed in the TWAIN.H file.

If the capability is supported and the operation is successful, it returns the Current, Default, and Available values. These values reflect previous MSG_SET operations on the capability which may have altered them from the TWAIN default values for the capability.

This operation may fail due to several causes. If the capability is not supported by the Source, the Return Code will be TWRC_FAILURE and the Condition Code will be set to TWCC_BADCAP.

MSG_SET

Changes the Current or Available Value(s) of the specified capability to those requested by the application. The application may choose to set just the capability's Current Value or it may specify a list of values for the Source to use as the complete set of Available Values for that capability.

Note - a Source is not required to limit values based on the application's request although it is recommended that they do so. If the Return Code indicates TWRC_FAILURE, check the Condition Code. A code of TWCC_BADVALUE can mean:

- The application sent an invalid value for this Source's range.
- The Source does not allow the setting of this capability.
- The Source doesn't allow the type of container used by the application to set this capability.

Capability negotiation gives the application developer power to guide the Source and control the images they receive from the Source. The negotiation typically occurs during State 4. The following material illustrates only one very basic capability and container structure. Refer to Chapter 4 for a more extensive discussion of capabilities including information on how to delay the negotiation of some capabilities beyond State 4.

Note: It is important here to once again remind application writers to always check the return code from any negotiated capabilities transactions.

Set the Capability to Specify the Number of Images the Application can Transfer

The capability that specifies how many images an application can receive during a TWAIN session is CAP_XFERCOUNT. All Sources must support this capability. Possible values for CAP_XFERCOUNT are:

Value:	Description:
1	Application wants to receive a single image.
greater than 1	Application wants to receive this specific number of images.
-1	Application can accept any arbitrary number of images during the session. **This is the default for this capability.**
0	This value has no legitimate meaning and the application should not set the capability to this value. If a Source receives this value during a MSG_SET operation, it should maintain the Current Value without change and return TWRC_FAILURE and TWCC_BADVALUE.

The default value allows multiple images to be transferred. The following is a simple code example illustrating the setting of a capability and specifically showing how to limit the number of images to one. Notice there are differences between the code for Windows and Macintosh applications. Both versions are included here with ifdef statements for MSWIN versus MAC.

```
TW_CAPABILITY   twCapability;
TW_INT16              count;
TW_STATUS             twStatus;
TW_UINT16             rc;
#ifdef _MSWIN_
pTW_ONEVALUE   pval;
#endif
#ifdef _MAC_
TW_HANDLE      h;
pTW_INT16      pInt16;
#endif

//-----Setup for MSG_SET for CAP_XFERCOUNT
twCapability.Cap = CAP_XFERCOUNT;
twCapability.ConType = TWON_ONEVALUE;

#ifdef _MSWIN_
twCapability.hContainer = GlobalAlloc(GHND, sizeof(TW_ONEVALUE));
pval = (pTW_ONEVALUE) GlobalLock(twCapability.hContainer);
pval->ItemType = TWTY_INT16;
pval->Item = 1;         //This app will only accept 1 image
GlobalUnlock(twCapability.hContainer);
#endif

#ifdef _MAC_
twCapability.hContainer = (TW_HANDLE)h = NewHandle(sizeof(TW_ONEVALUE));
((TW_ONEVALUE*)(*h))->ItemType = TWTY_INT16;
count = 1;         //This app will only accept 1 image
pInt16 = ((TW_ONEVALUE*)(*h))->Item;
*pInt16 = count;
#endif
```

```
//-----Set the CAP_XFERCOUNT
rc = (*pDSM_Entry) (&AppID,
                    &SourceID,
                    DG_CONTROL,
                    DAT_CAPABILITY,
                    MSG_SET,
                   (TW_MEMREF)&twCapability);

#ifdef _MSWIN_
GlobalFree((HANDLE)twContainer.hContainer);
#endif
#ifdef _MAC_
DisposHandle((HANDLE)twContainer.hContainer);
#endif

//-----Check Return Codes
//SUCCESS
if (rc == TWRC_SUCCESS)
    //the value was set
//APPROXIMATION MADE
else if (rc == TWRC_CHECKSTATUS)
    {
    //The value could not be matched exactly
    //MSG_GET to get the new current value
    twCapability.Cap = CAP_XFERCOUNT;
    twCapability.ConType = TWON_DONTCARE16;   //Source will specify
    twCapability.hContainer = NULL; //Source allocates and fills container

    rc = (*pDSM_Entry) (&AppID,
                &SourceID,
                DG_CONTROL,
                DAT_CAPABILITY,
                MSG_GET,
               (TW_MEMREF)&twCapability);

    //remember current value
    #ifdef _MSWIN_
    pval = (pTW_ONEVALUE) GlobalLock(twCapability.hContainer);
    count = pval->Item;
    //free hContainer allocated by Source
    GlobalFree((HANDLE)twCapability.hContainer);
    #endif

    #ifdef _MAC_
    pInt16 = ((TW_ONEVALUE*)(*h))->Item;
    count = *pInt16;
    //free hContainer allocated by Source
    DisposeHandle((HANDLE)twCapability.hContainer);
    #endif
    }
```

```
//MSG_SET FAILED
else if (rc == TWRC_FAILURE)
    {
    //check Condition Code
    rc = (*pDSM_Entry) (&AppID,
                &SourceID,
                DG_CONTROL,
                DAT_STATUS,
                MSG_GET,
               (TW_MEMREF)&twStatus);
    switch (twStatus.ConditionCode)
        {
        TWCC_BADCAP:
               //Source does not support setting this cap
               //All Sources must support CAP_XFERCOUNT
               break;
        TWCC_BADDEST:
               //The Source specified by pSourceID is not open
               break;
        TWCC_BADVALUE:
               //The value set was out of range for this Source
               //Use MSG_GET to determine what setting was made
               //See the TWRC_CHECKSTATUS case handled earlier
               break;
        TWCC_SEQERROR:
               //Operation invoked in invalid state
               break;
       }
    }
```

Other Capabilities

Image Type

Although not shown, the application should be aware of the Source's ICAP_PIXELTYPE and ICAP_BITDEPTH. If your application cannot accept all of the Source's Available Values, capability negotiation should be done. (Refer to Chapter 4.)

Transfer Mode

The default transfer mode is Native. That means the Source will access the largest block of memory available and use it to transfer the image to the application. The application does not need to do anything to select this transfer mode. If the application wishes to specify a different transfer mode, Disk File or Buffered Memory, further capability negotiation is required. (Refer to Chapter 4.)

State 4 to 5 - Request the Acquisition of Data from the Source

The Source device is open and capabilities have been negotiated. The application now enables the Source so it can show its user interface, if requested, and prepare to acquire data.

One Operation is Used:

DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Points to the Source's TW_IDENTITY structure.

pData

Points to a structure of type TW_USERINTERFACE.

The definition of TW_USERINTERFACE is:

```
typedef struct {
  TW_BOOL    ShowUI;
  TW_BOOL    ModalUI;
  TW_HANDLE  hParent;
} TW_USERINTERFACE, FAR *pTW_USERINTERFACE;
```

Set the ShowUI field to TRUE if you want the Source to display its user interface. Otherwise, set to FALSE.

The Source will set the ModalUI field to TRUE if its user interface is modal. If the interface is modeless, the field is set to FALSE.

The application sets the hParent field differently depending on the platform on which the application runs.

- **On Windows** - The application should place a handle to the Window that is acting as the Source's parent.
- **On Macintosh** - The application sets this field to NULL.

In response to the user choosing the application's Acquire menu option, the application sends this operation to the Source to enable it. The application typically requests that the Source display the Source's user interface to assist the user in acquiring data. If the Source is told to display its user interface, it will display it when it receives the operation triplet and it will set the ModalUI field of the data structure appropriately. Modal and Modaless interfaces are discussed in Chapters 4 and 5. Sources **must** check the `ShowUI` field and return an error if they cannot support the specified mode. In other words it is unacceptable for a source to ignore a `ShowUI = FALSE` request and still activate its user interface. The application may develop its own user interface instead of using the Source's. This is discussed in Chapter 4.

Note: Once the Source is enabled via the DG_CONTROL / DAT_USERINTERFACE/ MSG_ENABLEDS operation, all events that enter the application's main event loop must be immediately forwarded to the Source. The explanation for this was given earlier in this chapter when you were instructed to modify the event loop in preparation for a TWAIN session.

State 5 to 6 - Recognize that the Data Transfer is Ready

The Source is now working with the user to arrange the transfer of the desired data. Unlike all the earlier transitions, the Source, not the application, controls the transition from State 5 to State 6.

No Operations (from the application) are Used:

This transition is not triggered by the application sending an operation. The Source causes the transition.

Remember while the Source is enabled, the application is forwarding all events in its event loop to the Source by using the DG_CONTROL /DAT_EVENT / MSG_PROCESSEVENT operation. The TW_EVENT data structure associated with this operation looks like this:

```
typedef struct {
  TW_MEMREF  pEvent;    /*Windows pMSG or MAC pEvent                    */
  TW_UINT16  TWMessage; /*TW message from the Source to the application */
} TW_EVENT, FAR *pTW_EVENT;
```

The Source can set the TWMessage field to signal when the Source is ready to transfer data. Following each DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation, the application must check the TWMessage field. If it contains MSG_XFERREADY, the session is in State 6 and the Source will wait for the application to request the actual transfer of data.

State 6 to 7 - Start and Perform the Transfer

The Source indicated it is ready to transfer data. It is waiting for the application to inquire about the image details, initiate the actual transfer, and, hence, transition the session from State 6 to 7. If the initiation (DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET) fails, the session does not transition to State 7 but remains in State 6.

Two Operations are Used:

DG_IMAGE / DAT_IMAGEINFO / MSG_GET

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Points to the Source's TW_IDENTITY structure.

pData

Points to a structure of type TW_IMAGEINFO.

The definition of TW_IMAGEINFO is:

```
typedef struct {
   TW_FIX32    XResolution;
   TW_FIX32    YResolution;
   TW_INT32    ImageWidth;
   TW_INT32    ImageLength;
   TW_INT16    SamplesPerPixel;
   TW_INT16    BitsPerSample[8];
   TW_INT16    BitsPerPixel;
   TW_BOOL     Planar;
   TW_INT16    PixelType;
   TW_UINT32   Compression;
```

```
} TW_IMAGEINFO, FAR *pTW_IMAGEINFO;
```

The Source will fill in information about the image that is to be transferred. The application uses this operation to get the information regardless of which transfer mode (Native, Disk File, or Buffered Memory) will be used to transfer the data.

DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Points to the Source's TW_IDENTITY structure.

pData

Points to a TW_UINT32 variable. This is an exception from the typical pattern.

- **On Windows:** This is a pointer to a handle variable. For 16 bit MS Windows, the handle is stored in the low word of the 32-bit integer and the upper word is set to zero. If running under the WIN32 environment, this is a 32 bit window handle. The Source will set pHandle to point to a device-independent bitmap (DIB) that it allocates.
- **On Macintosh:** This is a pointer to a PicHandle. The Source will set pHandle to point to a PicHandle that the Source allocates.

In either case, the application is responsible for deallocating the memory block holding the Native-format image.

The application may want to inquire about the image data that it will be receiving. The DG_IMAGE / DAT_IMAGEINFO / MSG_GET operation allows this. Other operations, such as DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET, provide additional information. This information can be used to determine if the application actually wants to initiate the transfer.

To actually transfer the data in the Native mode, the application invokes the DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET operation. The Native mode is the default transfer mode and will be used unless a different mode was negotiated via capabilities in State 4. For the Native mode transfer, the application only invokes this operation once per image. The Source returns the TWRC_XFERDONE value when the transfer is complete. This type of transfer cannot be aborted by the application once initiated. (Whether it can be aborted from the Source's User Interface depends on the Source.) Use of the other transfer modes, Disk File and Buffered Memory, are discussed in Chapter 4.

The following code illustrates how to get information about the image that will be transferred and how to actually perform the transfer. This code segment is continued in the next section (State 7 to 6 to 5).

```
// After receiving MSG_XFERREADY
TW_UINT16 TransferNativeImage()
{
      TW_IMAGEINFO   twImageInfo;
      TW_UINT16      rc;
      TW_UINT32      hBitmap;
      TW_BOOL        PendingXfers = TRUE;
while (PendingXfers)
{
      rc = (*pDSM_Entry)(&AppId,
                              &SourceId,
```

```
                                DG_IMAGE,
                                DAT_IMAGEINFO,
                                MSG_GET,
                               (TW_MEMREF)&twImageInfo);
        if (rc == TWRC_SUCCESS)
               Examine the image information
               // Transfer the image natively
               hBitmap = NULL;
               rc = (*pDSM_Entry)(&AppId,
                                  SourceId,
                                  DG_IMAGE,
                                  DAT_IMAGENATIVEXFER,
                                  MSG_GET,
                                 (TW_MEMREF)&HbITMAP);
               // Check the return code
               switch(rc)
               {
                      case TWRC_XFERDONE:
        // Notes: hBitmap points to a valid image Native image (DIB or
        // PICT)
        // The application is now responsible for deallocating the memory.
        // The source is currently in state 7.
        // The application must now acknowledge the end of the transfer,
        // determine if other transfers are pending and shut down the data
        // source.
                      PendingXfers = DoEndXfer();  //Function found in code
                                                   //example in next section
                             break;
                      case TWRC_CANCEL:
         // The user canceled the transfer.
         // hBitmap is an invalid handle but memory was allocated.
         // Application is responsible for deallocating the memory.
         // The source is still in state 7.
         // The application must check for pending transfers and shut down
         // the data source.
                      PendingXfers = DoEndXfer();  //Function found in code
                                                   //example in next section
                             break;
                      case TWRC_FAILURE:
         // The transfer failed for some reason.
         // hBitmap is invalid and no memory was allocated.
         // Condition code will contain more information as to the cause of
         // the failure.
         // The state transition failed, the source is in state 6.
         // The image data is still pending.
         // The application should abort the transfer.
                        DoAbortXfer(MSG_RESET); //Function in next section
                        PendingXfers = FALSE;
                        break;
               }
         }
}
```

```
//Check the return code
switch (rc)
       {
       case TWRC_XFERDONE:
             //hBitMap points to a valid Native Image (DIB or PICT)
             //The application is responsible for deallocating the memory
             //The source is in State 7
             //Acknowledge the end of the transfer
                    goto LABEL_DO_ENDXFER //found in next section
              break;
       case TWRC_CANCEL:
              //The user canceled the transfer
              //hBitMap is invalid
              //The source is in State 7
              //Acknowledge the end of the transfer
                    goto LABEL_DO_ENDXFER //found in next section
              break;
       case TWRC_FAILURE:
              //The transfer failed
              //hBitMap is invalid and no memory was allocated
              //Check Condition Code for more information
              //The state transition failed, the source is in State 6
              //The image data is still pending
              //To abort the transfer
                    goto LABEL_DO_ENDXFER //found in code example for
                                          //the next section
              break;
   }
```

State 7 to 6 to 5 - Conclude the Transfer

While the transfer occurs, the session is in State 7. When the Source indicates via the Return Code that the transfer is done (TWRC_XFERDONE) or canceled (TWRC_CANCEL), the application needs to transition the session backwards.

One Operation is Used:

DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Points to the Source's TW_IDENTITY structure.

pData

Points to a structure of type TW_PENDINGXFERS.

The definition of TW_PENDINGXFERS is:

```
typedef struct {
  TW_UINT16  Count;
  TW_UINT32  Reserved;
} TW_PENDINGXFERS, FAR *pTW_PENDINGXFERS;
```

The DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation is sent by the application to the Source at the end of every transfer, successful or canceled, to indicate the application has received all the data it expected.

After this operation returns, the application should examine the pData->Count field to determine if there are more images waiting to be transferred. The value of pData->Count indicates the following:

Value	Description
pData->Count = 0	If zero, the Source will "automatically" transition back to State 5 without the application needing to take any additional action. **Application writers please make special note of this instance of an implied source transition.** The application should return to its main event loop and await notification from the Source (either MSG_XFERREADY or MSG_CLOSEDSREQ).
pData->Count = -1 or pData->Count > 0	The Source has more transfers available and is waiting in State 6. If the value is -1, that means the Source has another image available but it is unsure of how many more will be available. This might occur if the Source was controlling a device equipped with a document feeder and some unknown number of documents were stacked in that feeder. If the number of images is known, the Count will be a value greater than 0. Either way, the Source will remain in State 6 ready for the application to initiate another transfer. The Source will **NOT** send another MSG_XFERREADY to trigger this. The application should proceed as if it just received a MSG_XFERREADY.

If more images were pending and your application does not wish to transfer all of them, you can discard one or all pending images by doing the following:

- **To discard just the next pending image**, use the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation. Then, check the Count field again to determine if there are additional images pending.
- **To discard all pending images**, use the DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET operation. Following successful execution of this operation, the session will be in State 5.

The following code is a continuation of the code example started in the State 6 to 7 section. It illustrates how to conclude the transfer.

```
void DoEndXfer()
{
   TW_PENDINGXFERS      twPendingXfers;

   // If the return code from DG_IMAGE/DAT_IMAGENATIVEXFER/MSG_GET was
   // TWRC_CANCEL or TWRC_DONE

   // Acknowledge the end of the transfer
   rc = (*pDSM_Entry)(&AppId,
                                 SourceId,
                                 DG_CONTROL,
                                 DAT_PENDINGXFERS,
                                 MSG_ENDXFER,
                                 (TW_MEMREF)&twPendingXfers);

   if (rc == TWRC_SUCCESS)
   {
          // Check for additional pending xfers
          if (twPendingXfers.Count == 0)
          {
                 // Source is now in state 5. NOTE THE IMPLIED STATE
                 // TRANSITION! Disable and close the source and
                 // return to TransferNativeImage with a FALSE notifying
                 // it to not attempt further image transfers.

                 DisableAndCloseDS();
                 return(FALSE);
          }
          else
          {
                 // Source is in state 6 ready to transfer another image
                 if want to transfer this image
                 {
                        // returns to the caller, TransferNativeImage
                        // and allows the next image to transfer

                        return TRUE;
                 }
                 else if want to abort and skip over this transfer
                 {
                        // The current image will be skipped, and the
                        // next, if exists will be acquired by returning
                        // to TransferNativeImage

                        if (DoAbortXfer(MSG_ENDXFER) > 0)
                               return(TRUE);
                        else
                               return(FALSE);
                 }
                               }
          }
   }
}
```

```
TW_UINT16 DoAbortXfer(TW_UINT16 AbortType)
{
      rc = (*pDSM_Entry)(&AppId,
                                      SourceId,
                                      DG_CONTROL,
                                      DAT_PENDINGXFERS,
                                      MSG_ENDXFER,
                                     (TW_MEMREF)&twPendingXfers);
      if (rc == TWRC_SUCCESS)
      {
            // If the next image is to be skipped, but subsequent images
            // are still to be acquired, the PendingXfers will receive
            // the MSG_ENDXFER, otherwise, PendingXfers will receive
            // MSG_RESET.
            rc = (*pDSM_Entry)(&AppId,
                                         SourceId,
                                         DG_CONTROL,
                                         DAT_PENDINGXFERS,
                                         AbortType,
                                        (TW_MEMREF)&twPendingXfers);
      }
}
//To abort all pending transfers:
LABEL_ABORT_ALL:
            {
                rc = (*pDSM_Entry) (&AppID,
                       &SourceID,
                       DG_CONTROL,
                       DAT_PENDINGXFERS,
                       MSG_RESET,
                      (TW_MEMREF)&twPendingXfers);
                if (rc == TWRC_SUCCESS)
                       //Source is now in state 5
      }
}
```

State 5 to 1 - Disconnect the TWAIN Session

Once the application has acquired all desired data from the Source, the application can disconnect the TWAIN session. To do this, the application transitions the session backwards.

In the last section, the Source transitioned to State 5 when there were no more images to transfer (TW_PENDINGXFERS.Count = 0) or the application called the DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET operation to purge all remaining transfers. To back out the remainder of the session:

Three Operations (plus some platform-dependent code) are Used:

To move from State 5 to State 4

DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Points to the Source's TW_IDENTITY structure.

pData

Points to a structure of type TW_USERINTERFACE.

The definition of TW_USERINTERFACE is:

```
typedef struct {
  TW_BOOL    ShowUI;
  TW_BOOL    ModalUI;
  TW_HANDLE  hParent;
} TW_USERINTERFACE, FAR *pTW_USERINTERFACE;
```

Its contents are not used.

Note the following:

- **If the Source's User Interface was displayed:** This operation causes the Source's user interface, if displayed during the transition from State 4 to 5, to be lowered. This operation is sent by the application in response to a MSG_CLOSEDSREQ from the Source. This request from the Source appears in the TWMessage field of the TW_EVENT structure. It is sent back from the DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation used by the application to send events to the application.
- **If the application did not have the Source's User Interface displayed:** The application invokes this command when all transfers have been completed. In addition, the application could invoke this operation to transition back to State 4 if it wanted to modify one or more of the capability settings before acquiring more data.

To move from State 4 to State 3

DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Should reference a NULL value (indicates destination is Source Manager)

pData

Points to a structure of type TW_IDENTITY

This is the same TW_IDENTITY structure that you have used throughout the session to direct operation triplets to this Source.

When this operation is completed, the Source is closed. (In a more complicated scenario, if the application had more than one Source open, it must close them all before closing the Source Manager. Once all Sources are closed and the application does not plan to initiate any other TWAIN session with another Source, the Source Manager should be closed by the application.)

To move from State 3 to State 2

DG_CONTROL / DAT_PARENT / MSG_CLOSEDSM

pOrigin

Points to the application's TW_IDENTITY structure.

pDest

Should reference a NULL value (indicates destination is Source Manager)

pData

Typically, you would expect to see this point to a structure of type TW_PARENT but this is not the case. This is an exception to the usual situation where the DAT field of the triplet identifies the data structure for pData.

On Windows: pData points to the window handle (hWnd) that acted as the Source's "parent". The variable is of type TW_INT32. For 16 bit MS Windows, the handle is stored in the low word of the 32 bit integer and the upper word is set to zero. If running under the WIN32 environment, this is a 32 bit window handle.

On Macintosh: pData should be a 32-bit NULL value.

To Move from State 2 to State 1

Once the Source Manager has been closed, the application must unload the DLL (on Windows) or code resource (on Macintosh) from memory before continuing.

On Windows:

Use FreeLibrary(hDSMLib); where hDSMLib is the handle to the Source Manager DLL returned from the call to LoadLibrary( ) seen earlier (in the State 1 to 2 section).

On Macintosh:

```
HUnlock(DSMHandle);  /* unlock the handle to the Source Manager */
ReleaseResource(DSMHandle);  /* release the Source Manager      */
CloseResFile(DSMRefNum);     /* close the Source Manager file   */
```

TWAIN Session Review

Applications have flexibility regarding which state they leave their TWAIN sessions in between TWAIN commands (such as Select Source and Acquire).

For example:

- An application might load the Source Manager on start-up and unload it on exit. Or, it might load the Source Manager only when it is needed (as indicated by Select Source and Acquire).
- An application might open a Source and leave it in State 4 between acquires.

The following is the simplest view of application's TWAIN flow. All TWAIN actions are initiated by a TWAIN command, either user-initiated (Select Source and Acquire) or notification from the Source (MSG_XFERREADY and MSG_CLOSEDSREQ).

Application Receives	State	Application Action
Select Source...	1 -> 2	Load Source Manager
	2 -> 3	DG_CONTROL / DAT_PARENT / MSG_OPENDSM
		DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT
	3 -> 2	DG_CONTROL / DAT_PARENT / MSG_CLOSEDSM
	2 -> 1	Unload Source Manager
Acquire...	1 -> 2	Load Source Manager
	2 -> 3	DG_CONTROL / DAT_PARENT / MSG_OPENDSM
	3 -> 4	DG_CONTROL / DAT_IDENTITY / MSG_OPENDS
		Capability Negotiation
	4 -> 5	DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS
MSG_XFERREADY	6	For each pending transfer:
		DG_IMAGE / DAT_IMAGEINFO / MSG_GET
		DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
		DG_CONTROL / DAT_CAPABILITY / MSG_GETCURRENT
	6 -> 7	DG_IMAGE / DAT_IMAGExxxXFER / MSG_GET
	7 -> 6	DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER
	6 -> 5	Automatic transition to State 5 if TW_PENDINGXFERS.Count equals 0.
MSG_CLOSEDSREQ	5 -> 4	DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS
	4 -> 3	DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS
	3 -> 2	DG_CONTROL / DAT_PARENT / MSG_CLOSEDSM
	2 -> 1	Unload the Source Manager

Error Handling

Your application must be robust enough to recognize and handle error conditions that may occur during a TWAIN session. Every TWAIN operation triplet has a defined set of Return Codes and Conditions Codes that it may generate. These codes are listed on the reference pages for each triplet located in Chapter 7. Be sure to check the Return Code following every call to the DSM_Entry function. If it is TWRC_FAILURE, make sure your code checks the Condition Code and handles the error condition appropriately.

The following code segment illustrates the basic operations for doing this:

```
TW_STATUS    twStatus;
if (rc == TWRC_FAILURE)
       //check Condition Code
       rc = (*pDSM_Entry) (&AppID,
              &SourceID,
              DG_CONTROL,
              DAT_STATUS,
              MSG_GET,
             (TW_MEMREF)&twStatus);
       switch (twStatus.ConditionCode)
              //handle each possible Condition Code for the operation
```

Common Types of Error Conditions

Sequence Errors

The TWAIN protocol allows the invoking of specific operations only while the TWAIN session is in a particular state or states. The valid states for each operation are listed on the operation's reference pages in Chapter 7. If an operation is called from an inappropriate state, the call will return an error, TWRC_FAILURE, and set the Condition Code to TWCC_SEQERROR. Although this error should not occur if both the application and Source are behaving correctly, it is possible for the session to get out of sync.

If this error occurs, correct it by assuming the Source believes it is in State 7. The application should invoke the correct operations to back up from State 7 to State 6 and so on down the states until an operation succeeds. Then, the application can continue or terminate the session.

The following pseudo code illustrates this:

```
if (TWCC_SEQERROR)
      //  Assume State 7, start backing out from State 7 until
      //  the Condition Code != TWCC_SEQERROR
      State 7 to 6    DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER
      State 6 to 5    DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET
      State 5 to 4    DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS
      State 4 to 3    DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS
```

Low Memory Errors

Another common type of error condition occurs when insufficient memory is available to perform a requested operation. The most likely times for this to occur are:

- When a Source is being opened
- When a Source is being enabled
- During a Native image transfer

Your application must check the Return Code and Condition Code (TWRC_FAILURE / TWCC_LOWMEMORY) to recognize this. Your application may be able to free up sufficient memory to continue or it must quit.

State Transition Operation Triplet Errors

Many operations normally cause state transitions. If one of these operations fails, for example, returns TWRC_FAILURE, do not make the state transition. The application must check the Return Code following every operation and update the current state only if the operation succeeds.

An implied state transition during `DG_CONTROL/DAT_PENDINGXFERS/ MSG_ENDXFER` deserves special note here. If the `Count` field of the `TW_PENDINGXFERS` structure is zero then the source will automatically transition back to State 5. Application writers should be aware of this condition and react accordingly.

Requirements for an Application to be TWAIN-compliant

Applications are required to support only a subset of the defined TWAIN operations. As an application advances its need to set attributes it will also need to implement a more complete set of the defined operations. This includes provision of support for more transfer mechanisms.

An application **must** support the following to be considered TWAIN-compliant:

Operations

The following six operations are consumed by the Source Manager:

DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS

DG_CONTROL / DAT_IDENTITY / MSG_OPENDS

DG_CONTROL / DAT_PARENT / MSG_CLOSEDSM

DG_CONTROL / DAT_PARENT / MSG_OPENDSM

DG_CONTROL / DAT_STATUS / MSG_GET

The following seven operations are consumed by a Source:

DG_CONTROL / DAT_CAPABILITY / MSG_GET

DG_CONTROL / DAT_CAPABILITY / MSG_SET

DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT

DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET

DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER

DG_CONTROL / DAT_STATUS / MSG_GET

DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS

DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

Notices

Every application must support receipt of two notices from Sources. These are:

MSG_XFERREADY	indicates application can initiate transfer
MSG_CLOSEDSREQ	indicates the Source needs to be disabled

Capabilities

Applications must support one capability:

CAP_XFERCOUNT	Application sets the maximum number of transfers a Source is allowed to offer per session.

Applications that consume image information should support negotiation with the following capabilities:

ICAP_XFERMECH	the transfer mechanism to be used for the next transfer
ICAP_UNITS	unit of measure for all measured values (default is inches)
ICAP_PIXELTYPE	how image data is interpreted (Color, Gray, B&W, etc.)

Source requirements for TWAIN-compliance are presented in Chapter 5.

4

Advanced Application Implementation

Using TWAIN to acquire a raster image from a device is relatively simple to implement as demonstrated in Chapter 3. However, TWAIN also allows application developers to go beyond the simple acquisition of a single image in Native (DIB or PICT) format. These more advanced topics are discussed in this chapter. They include:

Chapter Contents

Capabilities

Capabilities, and the power of an application to negotiate capabilities with the Source, give control to TWAIN-compliant applications. In Chapter 3, you saw the negotiation of one capability, CAP_XFERCOUNT. This capability was negotiated during State 4 as is always the case unless delayed negotiation is agreed to by both the application and Source. In fact, there is much more to know about capabilities.

Capability Values

Several values are used to define each capability. As seen in Chapter 9, TWAIN defines a Default Value and a set of Allowed Values for each of the capabilities. The application is not able to modify the Default Value. However, it is able to limit the values offered to a user, the Available Values, to a subset of the Allowed Values and to select the capability's Current Value.

Default Value

When a Source is opened, the Current Values for each of its capabilities are set to the TWAIN Default Values listed in Chapter 9. If no default is defined by TWAIN, the Source will select a value for its default. An application can return a capability to its TWAIN-defined default by issuing a DG_CONTROL / DAT_CAPABILITY / MSG_RESET operation.

Although TWAIN defines defaults for many of the capabilities, a Source may have a different value that it would prefer to use as its default because it would be more efficient. For example, the Source may normally use a 0 bit in a black and white image to indicate white. However, the default for ICAP_PIXELFLAVOR is TWPF_CHOCOLATE which states that a 0 represents black. Although the TWAIN default is TWPF_CHOCOLATE, the Source's preferred default would be TWPF_VANILLA. When the application issues a DG_CONTROL / DAT_CAPABILITY / MSG_GETDEFAULT operation, the Source returns information about its preferred defaults. The Source and application may be able to negotiate a more efficient transfer based on this information.

Current Value

The application may request to set the Current Value of a capability. If the Source's user interface is displayed, the Current Value should be reflected (perhaps by highlighting). If the application sets the Current Value, it will be used for the acquire and transfer unless the user or an automatic Source process changes it. The application can determine if changes were made by checking the Current Value during State 6.

To determine just the capability's Current Value, use DG_CONTROL / DAT_CAPABILITY / MSG_GETCURRENT. To determine both the Current Value and the Available Values, use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation. For example, you could do a MSG_GET on ICAP_PIXELTYPE and the Source might return a TW_ENUMERATION container containing TWPT_BW, TWPT_GRAY, and TWPT_RGB as Available Values.

To set the Current Value:

Use DG_CONTROL / DAT_CAPABILITY / MSG_SET and one of the following containers:

- **TWON_ONEVALUE:** Place the desired value in TW_ONEVALUE.Item.
- **TWON_ARRAY:** Place only the desired items in TW_ARRAY.ItemList.

  These must be a subset of the items returned by the Source from a MSG_GET operation.

It is also possible to set Current Values using the TW_ENUMERATION and TW_RANGE containers. See the Available Values information for details.

Available Values

To limit the settings the Source can use during the acquire and transfer process, the application may be able to restrict the Available Values. The Source should not use a value outside these values. These restrictions should be reflected in the Source's user interface so unavailable values are not offered to the user.

For example, if the MSG_GET operation on ICAP_PIXELTYPE indicates the Source supports TWPT_BW, TWPT_GRAY, and TWPT_RGB images and the application only wants black and white images, it can request to limit the Available Values to black and white.

To limit the Available Values:

Use DG_CONTROL / DAT_CAPABILITY / MSG_SET and one of the following containers:

- **TWON_ENUMERATION:** Place only the desired values in the TW_ENUMERATION.ItemList field. The Current Value can also be set at this time by setting the CurrentIndex to point to the desired value in the ItemList.
- **TWON_RANGE:** Place only the desired values in the TW_RANGE fields. The current value can also be set by setting the CurrentValue field.

Note that TW_ONEVALUE and TW_ARRAY containers cannot be used to limit the Available Values.

Capability Negotiation

The negotiation process consists of three basic parts:

1. The application determines which capabilities a Source supports
2. The application sets the supported capabilities as desired
3. The application verifies that the settings were accepted by the Source

Negotiation (Part 1)
Application Determines Which Capabilities the Source Supports

Step 1

Application allocates a TW_CAPABILITY structure and fills its fields as follows:

- Cap = the CAP_ or ICAP_ name for the capability it is interested in
- ConType = TWON_DONTCARE16
- hContainer = NULL

Step 2

Application uses the TW_CAPABILITY structure in a DG_CONTROL / DAT_CAPABILITY / MSG_GET operation.

Step 3

The Source examines the Cap field to see if it supports the capability. If it does, it creates information for the application. In either case, it sets its Return Code appropriately.

Step 4

Application examines the Return Code, and maybe the Condition Code, from the operation.

If TWRC_SUCCESS then the Source does support the capability and

- The ConType field was filled by the Source with a container identifier (TWON_ARRAY, TWON_ENUMERATION, TWON_ONEVALUE, or TWON_RANGE)
- The Source allocated a container structure of ConType and referenced the hContainer field to this structure. It then filled the container with values describing the capability's Current Value, Default Value, and Available Values.

Based on the type of container and its contents (whose type is indicated by it ItemType field), the application can read the values. The application must deallocate the container.

If TWRC_FAILURE and TWCC_BADCAP

- Source does not support this capability

The application can repeat this process for every capability it wants to learn about. If the application really only wants to get the Current Value for a capability, it can use the MSG_GETCURRENT operation instead. In that case, the ConType will just be TWON_ONEVALUE or TWON_ARRAY but not TWON_RANGE or TWON_ENUMERATION.

Note: The capability, CAP_SUPPORTEDCAPS, returns a list of capabilities that a Source supports. But it doesn't indicate whether the supported capabilities can be negotiated, If the Source does not support the CAP_SUPPORTEDCAPS capabilities, it returns TWRC_FAILURE / TWCC_BADCAP.

Negotiation (Part 2)
The Application Sets the Supported Capability as Desired

Step 1

Application allocates a TW_CAPABILITY structure and fills its fields as follows:

- Cap = the CAP_ or ICAP_ name for the capability it is interested in
- ConType = TWON_ARRAY, TWON_ENUMERATION, TWON_ONEVALUE or TWON_RANGE (Refer to Chapter 9 to see each capability and what type(s) of container may be used to set a particular capability.)
- hContainer = The application must allocate a structure of type ConType and reference this field to it. (See the next step.)

Step 2

Application allocates a structure of type ConType and fills it. Based on values received from the Source during the MSG_GET, it can specify the desired Current Value and Available Values that it wants the Source to use. The application should not attempt to set the Source's Default Value, just put an appropriate constant in that field (ex. TWON_DONTCARE32).

Note: The application is responsible for deallocating the container structure when the operation is finished.

Step 3

Send the request to the Source using DG_CONTROL / DAT_CAPABILITY / MSG_SET.

Negotiation (Part 3)
The Application MUST Verify the Result of Their Request

Step 1

Even if a Source supports a particular capability, it is not required to support the setting of that capability. The application must examine the Return Code from the MSG_SET request to see what took place.

If TWRC_SUCCESS then the Source set the capability as requested.

If TWRC_CHECKSTATUS then

- The Source could not use one or more of your exact values. For instance, you asked for a value of 310 but it could only accept 100, 200, 300, or 400. Your request was within its legitimate range so it rounded it to its closest valid setting.

Use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation to determine the current and available settings at this time. This is the only way to determine if the Source's choice was acceptable to your application.

If TWRC_FAILURE / TWCC_BADVALUE then

- Either the Source is not granting your request to set or restrict the value.
- Or, your requested values were not within its range of legitimate values. It may have attempted to set the value to its closest available value.

Use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation to determine the current and available settings at this time. This is the only way to determine if your application can continue without your requested values.

You can repeat the setting and verifying processes for every capability of interest to your application. Remember, your application must deallocate all container structures.

The Most Common Capabilities

TWAIN defines over fifty capabilities. Although the number may seem overwhelming, it is easier to handle if you recognize that some of the capabilities are more commonly used. Here are some of these capabilities:

Basic Capabilities

Units

The ICAP_UNITS capability determines the unit of measure which will be used by the Source. The default is inches but centimeters, pixels, etc. are allowed. This capability's value is used when measuring several other values in capabilities and data structures including:

ICAP_PHYSICALHEIGHT,

ICAP_PHYSICALWIDTH,

ICAP_XNATIVERESOLUTION,

ICAP_YNATIVERESOLUTION,

ICAP_XRESOLUTION,

ICAP_YRESOLUTION,

TW_FRAME,

TW_IMAGEINFO.XResolution,

TW_IMAGEINFO.YResolution

Sense of the Pixel

The ICAP_PIXELFLAVOR specifies how a bit of data should be interpreted when transferred from Source to application. The default is TWPF_CHOCOLATE which means a 0 indicates black (or the darkest color). The alternative, TWPF_VANILLA, means a 0 indicates white (or the lightest color).

Resolution

The image resolution is reported in the TW_IMAGEINFO structure. To inquire or set the Source's resolution, use ICAP_XRESOLUTION and ICAP_YRESOLUTION.

Refer also to ICAP_XNATIVERESOLUTION and ICAP_YNATIVERESOLUTION.

Image Type Capabilities

Types of Pixel

The application should negotiate ICAP_PIXELTYPE and ICAP_BITDEPTH unless it can handle all pixel types at all bit depths. The allowed pixel types are: TWPT_BW, TWPT_GRAY, TWPT_RGB, TWPT_PALETTE, TWPT_CMY, TWPT_CMYK, TWPT_YUV, TWPT_YUVK, and TWPT_CIEXYZ.

Depth of the Pixels (in bits)

A pixel type such as TWPT_BW allows only 1 bit per pixel (either black or white). The other pixel types may allow a variety of bits per pixel (4-bit or 8-bit gray, 8-bit or 24-bit color). Be sure to set the ICAP_PIXELTYPE first, then set the ICAP_BITDEPTH.

Parameters for Acquiring the Image

Exposure

Several capabilities can influence this. They include ICAP_BRIGHTNESS, ICAP_CONTRAST, ICAP_SHADOW, ICAP_HIGHLIGHT, ICAP_GAMMA, and ICAP_AUTOBRIGHT.

Scaling

To instruct a Source to scale an image before transfer, refer to ICAP_XSCALING and ICAP_YSCALING.

Rotation

To instruct a Source to rotate the image before transfer, refer to ICAP_ROTATION and ICAP_ORIENTATION.

Capability Containers in Code Form

Capability information is passed between application and Source by using data structures called containers: TW_ARRAY, TW_ENUMERATION, TW_ONEVALUE, and TW_RANGE. The actions needed to create (pack) and read (unpack) containers are illustrated here in the following code segments. Containers are flexible in that they can be defined to contain one of many types of data. Only one ItemType (TWTY_xxx) is illustrated per Container (TWON_xxx) here. Refer to the toolkit disk for complete packing and unpacking utilities that you can use with containers.

Reading (unpacking) a Container from a MSG_GET Operation

```
//----------------------------------------------
//Example of DG_CONTROL / DAT_CAPABILITY / MSG_GET
//----------------------------------------------
TW_CAPABILITY   twCapability;
TW_INT16         rc;

//Setup TW_CAPABILITY Structure
   twCapability.Cap = Cap;     //Fill in capability of interest
   twCapability.ConType = TWON_DONTCARE16;
   twCapability.hContainer = NULL;

//Send the Triplet
   rc = (*pDSM_Entry)(&AppID,
                &SourceID,
                DG_CONTROL,
                DAT_CAPABILITY,
                MSG_GET,
               (TW_MEMREF)&twCapability);
```

```
//Check return code
   if (rc == TWRC_SUCCESS)
{
//Switch on Container Type
       switch (twCapability.ConType)
       {
//-----ENUMERATION
          case TWON_ENUMERATION:
          {
          pTW_ENUMERATION   pvalEnum;
          TW_UINT16         valueU16;
          TW_UINT16         index;
          pvalEnum = (pTW_ENUMERATION)GlobalLock(twCapability.hContainer);
          NumItems = pvalEnum->NumItems;
          CurrentIndex = pvalEnum->CurrentIndex;
          DefaultIndex = pvalEnum->DefaultIndex;

          for (index = 0; index < pvalEnum->NumItems; index++)
          {
              if (pvalEnum->ItemType == TWTY_UINT16)
              {
                  valueU16 = ((TW_UINT16)(pvalEnum->ItemList[index*2]));
                  //Store Item Value
              }
          }
          GlobalUnlock(twCapability.hContainer);
          }
          break;

//-----ONEVALUE
          case TWON_ONEVALUE:
          {
          pTW_ONEVALUE      pvalOneValue;
          TW_BOOL         valueBool;

         pvalOneValue = (pTW_ONEVALUE)GlobalLock(twCapability.hContainer);
         if (pvalOneValue->ItemType == TWTY_BOOL)
         {
               valueBool = (TW_BOOL)pvalOneValue->Item;
               //Store Item Value
         }
         GlobalUnlock(twCapability.hContainer);
         }
         break;
```

```
//-----RANGE
          case TWON_RANGE:
          {
          pTW_RANGE         pvalRange;
          pTW_FIX32         pTWFix32;
          float            valueF32;
          TW_UINT16         index;

            pvalRange = (pTW_RANGE)GlobalLock(twCapability.hContainer);

            if ((TW_UINT16)pvalRange->ItemType == TWTY_FIX32)
            {
               pTWFix32 = &(pvalRange->MinValue);
               valueF32 = FIX32ToFloat(*pTWFix32);
               //Store Item Value

               pTWFix32 = &(pvalRange->MaxValue);
               valueF32 = FIX32ToFloat(*pTWFix32);
               //Store Item Value

               pTWFix32 = &(pvalRange->StepSize);
               valueF32 = FIX32ToFloat(*pTWFix32);
               //Store Item Value

            }
            GlobalUnlock(twCapability.hContainer);
         }
         break;
//-----ARRAY
          case TWON_ARRAY:
          {
          pTW_ARRAY         pvalArray;
          TW_UINT16         valueU16;
          TW_UINT16         index;

            pvalArray = (pTW_ARRAY)GlobalLock(twCapability.hContainer);

            for (index = 0; index < pvalArray->NumItems; index++)
            {
               if (pvalArray->ItemType == TWTY_UINT16)
               {
                  valueU16 = ((TW_UINT16)(pvalArray->ItemList[index*2]));
                  //Store Item Value
               }
            }
            GlobalUnlock(twCapability.hContainer);
         }
         break;
      }   //End Switch Statement

     GlobalFree(twCapability.hContainer);

   } else {
      //Capability MSG_GET Failed check Condition Code
   }
```

```
/*********************************************************
* Fix32ToFloat
* Convert a FIX32 value into a floating point value.
*********************************************************/

float FIX32ToFloat (TW_FIX32    fix32)
{
   float    floater;

   floater = (float)fix32.Whole + (float)fix32.Frac / 65536.0;
   return floater;
}
```

Creating (packing) a Container for a MSG_SET Operation

```
//----------------------------------------------
//Example of DG_CONTROL / DAT_CAPABILITY / MSG_SET
//----------------------------------------------
TW_CAPABILITY   twCapability;
TW_INT16         rc;
TW_UINT32         NumberOfItems;

   twCapability.Cap = Cap;      //Insert Capability of Interest
   twCapability.ConType = Container;
      //Use TWON_ONEVALUE or TWON_ARRAY to set current value
      //Use TWON_ENUMERATION or TWON_RANGE to limit available values

   switch (twCapability.ConType)
   {
//-----ENUMERATION
      case TWON_ENUMERATION:
      {
      pTW_ENUMERATION   pvalEnum;

         //The number of Items in the ItemList
         NumberOfItems = 2;


       //Allocate memory for the container and additional ItemList
       // entries
       twCapability.hContainer = GlobalAlloc(GHND,
          (sizeof(TW_ENUMERATION) + sizeof(TW_UINT16) * (NumberOfItems)));
       pvalEnum = (pTW_ENUMERATION)GlobalLock(twCapability.hContainer);

       pvalEnum->NumItems = 2      //Number of Items in ItemList
       pvalEnum->ItemType = TWTY_UINT16;
       ((TW_UINT16)(pvalEnum->ItemList[0])) = 1;
       ((TW_UINT16)(pvalEnum->ItemList[1])) = 2;

       GlobalUnlock(twCapability.hContainer);
    }
    break;
```

```
//-----ONEVALUE
      case TWON_ONEVALUE:
      {
      pTW_ONEVALUE      pvalOneValue;

         twCapability.hContainer = GlobalAlloc(GHND, sizeof(TW_ONEVALUE));
         pvalOneValue = (pTW_ONEVALUE)GlobalLock(twCapability.hContainer);

         (TW_UINT16)pvalOneValue->ItemType = TWTY_UINT16;
         (TW_UINT16)pvalOneValue->Item = 1;

         GlobalUnlock(twCapability.hContainer);
      }
      break;

//-----RANGE
      case TWON_RANGE:
      {
      pTW_RANGE         pvalRange;
      TW_FIX32         TWFix32;
      float            valueF32;

         twCapability.hContainer = GlobalAlloc(GHND, sizeof(TW_RANGE));
         pvalRange = (pTW_RANGE)GlobalLock(twCapability.hContainer);

         (TW_UINT16)pvalRange->ItemType = TWTY_FIX32;
         valueF32 = 100;
         TWFix32 = FloatToFIX32 (valueF32);
         pvalRange->MinValue = *((pTW_INT32) &TWFix32);
         valueF32 = 200;
         TWFix32 = FloatToFIX32 (valueF32);
         pvalRange->MaxValue = *((pTW_INT32) &TWFix32);

         GlobalUnlock(twCapability.hContainer);
      }
      break;

//-----ARRAY
      case TWON_ARRAY:
      {
      pTW_ARRAY         pvalArray;

         //The number of Items in the ItemList
         NumberOfItems = 2;

       //Allocate memory for the container and additional ItemList entries
       twCapability.hContainer = GlobalAlloc(GHND,
         (sizeof(TW_ARRAY) + sizeof(TW_UINT16) * (NumberOfItems)));
       pvalArray = (pTW_ARRAY)GlobalLock(twCapability.hContainer);

         (TW_UINT16)pvalArray->ItemType = TWTY_UINT16;
         (TW_UINT16)pvalArray->NumItems = 2;
         ((TW_UINT16)(pvalArray->ItemList[0])) = 1;
         ((TW_UINT16)(pvalArray->ItemList[1])) = 2;

         GlobalUnlock(twCapability.hContainer);
      }
      break;
   }
```

```
//-----MSG_SET
   rc = (*pDSM_Entry)(&AppID,
                  &SourceID,
                  DG_CONTROL,
                  DAT_CAPABILITY,
                  MSG_SET,
                  (TW_MEMREF)&twCapability);
   GlobalFree(twCapability.hContainer);
   switch (rc)
   {
      case TWRC_SUCCESS:
         //Capability's Current or Available value was set as specified
      case TWRC_CHECKSTATUS:
        //The Source matched the specified value(s) as closely as possible
         //Do a MSG_GET to determine the settings made
      case TWRC_FAILURE:
         //Check the Condition Code for more information
   }
/*********************************************************
* FloatToFix32
* Convert a floating point value into a FIX32.
*********************************************************/
TW_FIX32 FloatToFix32 (float floater)
{
   TW_FIX32 Fix32_value;
   TW_INT32 value = (TW_INT32) (floater * 65536.0 + 0.5);
   Fix32_value.Whole = value >> 16;
   Fix32_value.Frac = value & 0x0000ffffL;
   return (Fix32_value);
}
```

Delayed Negotiation - Negotiating Capabilities After State 4

Applications may inquire about a Source's capability values at any time during the session with the Source. However, as a rule, applications can only request to **set** a capability during State 4. The rationale behind this restriction is tied to the display of the Source's user interface when the Source is enabled. Many Sources will modify the contents of their user interface in response to some of the application's requested settings. These user interface modifications prevent the user from selecting choices that do not meet the application's requested values. The Source's user interface is never displayed in State 4 so changes can be made without the user's awareness. However, the interface may be displayed in States 5 through 7.

Some capabilities have no impact on the Source's user interface and the application may really want to set them later than State 4. To allow delayed negotiation, the application must request, during State 4, that a particular capability be able to be set later (during States 5 or 6). The Source may agree to this request or deny it. The request is negotiated by the application with the Source by using the DG_CONTROL / DAT_CAPABILITY operations on the CAP_EXTENDEDCAPS capability.

On the CAP_EXTENDEDCAPS capability, the DG_CONTROL / DAT_CAPABILITY operations:

MSG_GET

Indicates all capabilities that the Source is willing to negotiate in State 5 or 6.

MSG_SET

Specifies which capabilities the application wishes to negotiate in States 5 or 6.

MSG_GETCURRENT

Provides a list of all capabilities which the Source and application have agreed to allow to be negotiated in States 5 or 6.

As with any other capability, if the Source does not support negotiating CAP_EXTENDEDCAPS, it will return the Return Code TWRC_FAILURE with the Condition Code TWCC_BADCAP.

If an application attempts to set a capability in State 5 or 6 and the Source has not previously agreed to this arrangement, the operation will fail with a Return Code of TWRC_FAILURE and a Condition Code of TWCC_SEQERROR.

If an application does not use the Source's user interface but presents its own, the application controls the state of the Source explicitly. If the application wants to set the value of any capability, it returns the Source to State 4 and does so. Therefore, an application using its own user interface will probably not need to use CAP_EXTENDEDCAPS.

Options for Transferring Data

As discussed previously, there are three modes defined by TWAIN for transferring data:

- Native
- Disk File
- Buffered Memory

A Source is required to support Native and Buffered Memory transfers.

Native Mode Transfer

The use of Native mode, the default mode, for transferring data was covered in Chapter 3. There is one potential limitation that can occur in a Native mode transfer. That is, there may not be an adequately large block of RAM available to hold the image. This situation will not be discovered until the transfer is attempted when the application issues the DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET operation.

When the lack of memory appears, the Source may respond in one of several ways. It can:

- Simply fail the operation.
- Clip the image to make it fit in the available RAM - The Source should notify the user that the clipping operation is taking place due to limited RAM. The clipping should maintain both the aspect ratio of the selected image and the origin (upper-left).
- Interact with the user to allow them to resize the image or cancel the capture.

The Return Code / Condition Code returned from the DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET operation may indicate one of these actions occurred.

If the Return Code is TWRC_XFERDONE:

This indicates the transfer was completed and the session is in State 7. However, it does not guarantee that the Source did not clip the image to make it fit. Even if the application issued a DG_IMAGE / DAT_IMAGEINFO / MSG_GET operation prior to the transfer to determine the image size, it cannot assume that the ImageWidth and ImageLength values returned from that operation really apply to the image that was ultimately transferred. If the dimensions of the image are important to the application, the application should always check the actual transferred image size after the transfer is completed. To do this:

1. Execute a DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation to move the session from State 7 to State 6 (or 5).
2. Determine the actual size of the image that was transferred:
   a. **On Windows** - Read the DIB header
   b. **On Macintosh** - Check the picFrame in the Picture

If the Return Code is TWRC_CANCEL:

The acquisition was canceled by the user. The session is in State 7. Execute a DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation to move the session from State 7 to State 6 (or 5).

If the Return Code is TWRC_FAILURE:

Check the Condition Code to determine the cause of the failure. The session is in State 6. No memory was allocated for the DIB or PICT. The image is still pending. If lack of memory was the cause, you can try to free additional memory or discard the pending image by executing DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER.

Disk File Mode Transfer

Determine if a Source Supports the Disk File Mode

- Use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation.
- Set the TW_CAPABILITY's Cap field to ICAP_XFERMECH.
- The Source returns information about the transfer modes it supports in the container structure pointed to by the hContainer field of the TW_CAPABILITY structure. The disk file mode is identified as TWSX_FILE. Sources are not required to support Disk File Transfer so it is important to verify its support.

After Verifying Disk File Transfer is Supported, Set Up the Transfer

During State 4:

- Set the ICAP_XFERMECH to TWSX_FILE. Use the DG_CONTROL / DAT_CAPABILITY / MSG_SET operation.
- Use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation to determine which file formats the Source can support. Set TW_CAPABILITY.Cap to ICAP_IMAGEFILEFORMAT and execute the MSG_GET. The Source returns the supported format identifiers which start with TWFF_ and may include TWFF_PICT, TWFF_BMP, TWFF_TIFF, etc. They are listed in the TWAIN.H file and in the Constants section of Chapter 8.

During States 4, 5, or 6:

To set up the transfer, the DG_CONTROL / DAT_SETUPFILEXFER operations of MSG_GET, MSG_GETDEFAULT, and MSG_SET can be used. The data structure used in the DSM_Entry call is a TW_SETUPFILEXFER structure:

```
typedef struct {
  TW_STR255   FileName; /* File to contain data       */
  TW_UINT16   Format;   /* A TWFF_xxx constant        */
  TW_HANDLE   VRefNum;  /* Used for Macintosh only    */
} TW_SETUPFILEXFER, FAR *pTW_SETUPFILEXFER;
```

The application could use the MSG_GETDEFAULT operation to determine the default file format and filename (TWAIN.TMP in the current directory). If acceptable, the application could just use that file. However, most applications prefer to set their own values for filename and format. The MSG_SET operation allows this. It is done during State 6. To set your own filename and format, do the following:

1. Create the file specified for the transfer and close it. Be sure the Source has permission to read and write this file.
2. Allocate the required TW_SETUPFILEXFER structure. Then, fill in these fields:

   FileName - the desired file name. On Windows, be sure to include the complete path name.

   Format - the constant for the desired, and supported, format. (TWFF_xxxx). If you set it to an unsupported format, the operation returns TWRC_FAILURE / TWCC_BADVALUE and the Source resets itself to write data to its default file.

   VRefNum - On Macintosh, write the file's volume reference number. On Windows, fill the field with a TWON_DONTCARE16.
3. Invoke the DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET operation.

Execute the Transfer into the File

After the application receives the MSG_XFERREADY notice from the Source and has issued the DG_CONTROL / DAT_SETUPFILEXFER / MSG_GET operation:

Use the following operation: DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET

This operation does not have an associated data structure but just uses NULL for the pData parameter in the DSM_Entry call.

- **If the application has not specified a filename (during the setup)** - the Source will use either its default file or the last file information it was given.
- **If the file specified by the application does not exist** - the Source should create it.
- **If the file exists but already has data in it** - the Source should overwrite the existing data. Notice, if you are transferring multiple files and using the same file name each time, you will overwrite the data unless you copy it to a different filename between transfers.

Note: The application cannot abort a Disk File transfer once initiated. However, the Source's user interface may allow the user to cancel the transfer.

Following execution, be sure to check the Return Code:

TWRC_XFERDONE = File was written successfully. The application needs to invoke the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER to transition the session back to State 6 (or 5) as was illustrated in Chapter 3.

TWRC_CANCEL = The user canceled the transfer. The contents of the file are undefined. Invoke DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER to transition the session back to State 6 (or 5) as was illustrated in Chapter 3.

TWRC_FAILURE
The Source remained in State 6.
The contents of the file are undefined.
The image is still pending. To discard it, use DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER.

Check the Condition Code to determine the cause of the failures. The alternatives are:

TWCC_BADDEST = Operation aimed at invalid Source

TWCC_OPERATIONERROR = Either the file existed but could not be accessed or a system error occurred during the writing

TWCC_SEQERROR = Operation invoked in invalid state (i.e. not 6)

Buffered Memory Mode Transfer

Set Capability Values for the Buffered Memory Mode, if Desired

Data is typically transferred in uncompressed format. However, if you are interested in knowing if the Source can transfer compressed data when using the buffered memory mode, perform a DG_CONTROL / DAT_CAPABILITY / MSG_GET on the ICAP_COMPRESSION. The values will include TWCP_NONE (the default) and perhaps others such as TWCP_PACKBITS, TWCP_JPEG ,etc. (See the list in the Constants section of Chapter 8.) More information on compression is available later in this chapter in the section called Transfer of Compressed Data.

Set up the Transfer

During State 4:

Set the ICAP_XFERMECH to TWSX_MEMORY by using the DG_CONTROL / DAT_CAPABILITY / MSG_SET operation.

During States 4, 5, or 6:

The DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET operation is used by the application to determine what buffer sizes the Source wants to use during the transfer. The Source might have more accurate information in State 6.

The data structure used in the DSM_Entry call is a TW_SETUPMEMXFER structure:

```
typedef struct {
  TW_UINT32   MinBufSize  /* Minimum buffer size in bytes   */
  TW_UINT32   MaxBufSize  /* Maximum buffer size in bytes   */
  TW_UINT32   Preferred   /* Preferred buffer size in bytes */
} TW_SETUPMEMXFER, FAR *pTW_SETUPMEMXFER;
```

The Source will fill in the appropriate values for its device.

Buffers Used for Uncompressed Strip Transfers

- The application is responsible for allocating and deallocating all memory used during the buffered memory transfer.
- For optimal performance, create buffers of the Preferred size.
- In all cases, the size of the allocated buffers must be within the limits of MinBufSize to MaxBufSize. If outside of these limits, the Source will fail the transfer operation with a Return Code of TWRC_FAILURE / TWCC_BADVALUE.
- If using more than one buffer, all buffers must be the same size.
- Raster lines must be double-word aligned and padded with zeros is recommended .

Execute the Transfer Using Buffers

After the application receives the MSG_XFERREADY notice from the Source and has issued the DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET operation:

- Allocate one or more buffers of the same size. The best size is the one indicated by the TW_SETUPMEMXFER.Preferred field. If that is impossible, be certain the buffer size is between MinBufSize and MaxBufSize.
- Allocate the TW_IMAGEMEMXFER structure. It will be used in the DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET operation.

The TW_IMAGEMEMXFER structure looks like this:

```
typedef struct {
  TW_UINT16  Compression;
  TW_UINT32  BytesPerRow;
  TW_UINT32  Columns;
  TW_UINT32  Rows;
  TW_UINT32  XOffset;
  TW_UINT32  YOffset;
  TW_UINT32  BytesWritten;
  TW_MEMORY  Memory;
} TW_IMAGEMEMXFER, FAR *pTW_IMAGEMEMXFER;
```

Fill in the TW_IMAGEMEMXFER's first field with TWON_DONTCARE16 and the following six fields with TWON_DONTCARE32.

The TW_MEMORY structure embedded in there looks like this:

```
typedef struct {
  TW_UINT32  Flags;
  TW_UINT32  Length;
  TW_MEMREF  TheMem;
} TW_MEMORY, FAR *pTW_MEMORY;
```

Fill in the TW_MEMORY structure as follows:

Memory.Flags

Place TWMF_APPOWNS bit-wise ORed with TWMF_POINTER or TWMF_HANDLE

Memory.Length

The size of the buffer in bytes

Memory.TheMem

A handle or pointer to the memory buffer allocated above (depending on which one was specified in the Flags field).

Following each buffer transfer, the Source will have filled in all the fields except Memory which it uses as a reference to the memory block for the data.

The flow of the transfer of buffers is as follows:

Step 1

Buffered Memory transfers provide no embedded header information. Therefore, the application must determine the image attributes. After receiving the MSG_XFERREADY, i.e. while in State 6, the application issues the DG_IMAGE / DAT_IMAGEINFO / MSG_GET and DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET operations to learn about the image's bitmap characteristics and the size and location of the original image on the original page (before scaling or other processing). If additional information is desired, use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation.

Step 2

The application issues DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET.

Step 3

The application checks the Return Code.

If TWRC_SUCCESS:

Examine the TW_IMAGEMEMXFER structure for information about the buffer. If you plan to reuse the buffer, copy the data to another location.

Loop back to Step 2 to get another buffer. Be sure to reinitialize the information in the TW_IMAGEMEMXFER structure (including the Memory fields), if necessary. Issue another DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET operation.

If TWRC_XFERDONE:

This is how the Source indicates it just transferred the last buffer successfully. Examine the TW_IMAGEMEMXFER structure for information about the buffer. Perhaps, copy the data to another location, as desired, then go to Step 4.

If TWRC_CANCEL:

The user aborted the transfer. The application must send a DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER as described in Chapter 3 to move from State 7 to State 6 (or 5).

If TWRC_FAILURE:

Examine the Condition Code to determine the cause and handle it.
If the failure occurred during the transfer of the first buffer, the session is in State 6. If the failure occurred on a subsequent buffer, the session is in State 7.

The contents of the buffer are invalid and the transfer of the buffer is still pending. To abort it, use DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER.

Step 4

Once the TWRC_XFERDONE has been returned, the application must send the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER to conclude the transfer. This was described in Chapter 3 in the section called State 7 to 6 to 5 - Conclude the Transfer.

Note: The majority of Sources divide the image data into strips when using buffered transfers. A strip is a horizontal band starting at the leftmost side of the image and spanning the entire width but covering just a portion of the image length. The application can verify that strips are being used if the information returned from the Source in the TW_IMAGEMEMXFER structure's XOffset field is zero and the Columns field is equal to the value in the TW_IMAGEINFO structure's ImageWidth field.

An alternative to strips is the use of tiles although they are used by very few Sources. Refer to the TW_IMAGEMEMXFER information in Chapter 8 for an illustration of tiles.

The Image Data and Its Layout

The image which is transferred from the Source to the application has several attributes. Some attributes describe the size of the image. Some describe where the image was located on the original page. Still others might describe information such as resolution or number of bits per pixel. TWAIN provides means for the application to learn about these attributes.

Users are often able to select and modify an image's attributes through the Source's user interface. Additionally, TWAIN provides capabilities and operations that allow the application to impact these attributes prior to acquisition and transfer.

Getting Information About the Image That will be Transferred

Before the transfer occurs, while in State 6, the Source can provide information to the application about the actual image that it is about to transfer. Note, the information is lost once the transfer takes place so the application should save it, if needed. This information can be retrieved through two operations:

- DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
- DG_IMAGE / DAT_IMAGEINFO / MSG_GET

The area of an image to be acquired will always be a rectangle called a frame. There may be one or more frames located on a page. Frames can be selected by the user or designated by the application. The TW_IMAGELAYOUT structure communicates where the image was located

on the original page relative to the origin of the page. It also indicates, in its FrameNumber field, if this is the first frame, or a later frame, to be acquired from the page.

The TW_IMAGELAYOUT structure looks like this:

```
typedef struct {
   TW_FRAME      Frame;
   TW_UINT32     DocumentNumber;
   TW_UINT32     PageNumber;
   TW_UINT32     FrameNumber;
   } TW_IMAGELAYOUT, FAR *pTW_IMAGELAYOUT;
```

The TW_FRAME structure specifies the values for the Left, Right, Top, and Bottom of the frame to be acquired. Values are given in ICAP_UNITS.

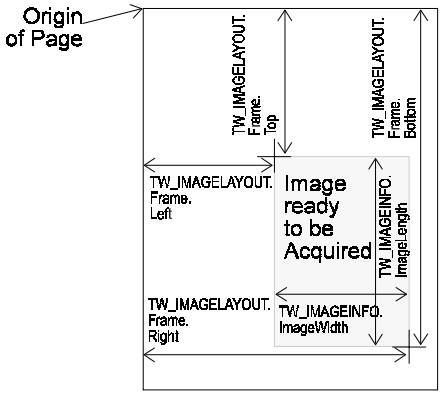


Figure 4-1. TW_FRAME Structure

The DG_IMAGE / DAT_IMAGEINFO / MSG_GET operation communicates other attributes of the image being transferred. The TW_IMAGEINFO structure looks like this:

```
typedef struct {
   TW_FIX32    XResolution;
   TW_FIX32    YResolution;
   TW_INT32    ImageWidth;
   TW_INT32    ImageLength;
   TW_INT16    SamplesPerPixel;
   TW_INT16    BitsPerSample[8];
   TW_INT16    BitsPerPixel;
   TW_BOOL     Planar;
   TW_INT16    PixelType;
   TW_UINT16   Compression;
   } TW_IMAGEINFO, FAR * pTW_IMAGEINFO;
```

Notice how the ImageWidth and ImageLength relate to the frame described by the TW_IMAGELAYOUT structure.

Changing the Image Attributes

Normally, the user will select the desired attributes. However, the application may wish to do this initially during State 4. For example, if the user interface will not be displayed, the application may wish to select the frame. The application can use a DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET operation to define the area (frame) to be acquired. Although, there is no corresponding DG_IMAGE / DAT_IMAGEINFO / MSG_SET operation, the application can change those attributes by setting capabilities and the TW_IMAGELAYOUT data structure.

Here are the relationships:

TW_IMAGEINFO fields	Capability or data structure that impacts the attribute
XResolution	ICAP_XRESOLUTION
YResolution	ICAP_YRESOLUTION
ImageWidth	TW_IMAGELAYOUT.TW_FRAME.Right - TW_FRAME.Left **
ImageLength	TW_IMAGELAYOUT.TW_FRAME.Bottom - TW_FRAME.Top **
SamplesPerPixel	ICAP_PIXELTYPE (i.e. TWPT_BW has 1, TWPT_RGB has 3)
BitsPerSample	Calculated by BitsPerPixel divided by SamplesPerPixel
BitsPerPixel	ICAP_BITDEPTH
Planar	ICAP_PLANARCHUNKY
PixelType	ICAP_PIXELTYPE
Compression	ICAP_COMPRESSION

** ImageWidth and ImageLength are actually provided in pixels whereas TW_FRAME uses ICAP_UNITS.

Transfer of Multiple Images

Chapter 3 discussed the transfer of a single image. Transferring multiple images simply requires looping through the single-image transfer process repeatedly whenever more images are available. Two classes of issues arise when considering multiple image transfer under TWAIN:

- What state transitions are allowable when a session is at an inter-image boundary?
- What facilities are available to support the operation of a document feeder? This includes issues related to high-performance scanning.

This section starts with a review of the single-image transfer process. This is followed by a discussion of options available to an application once the transfer of a single image is complete. Finally, document feeder issues are presented.

To briefly review the single-image transfer process:

- The application enables the Source and the session moves from State 4 to State 5.
- The Source sends the application a MSG_XFERREADY when an image is ready for transfer.
- The application uses DG_IMAGE / DAT_IMAGEINFO / MSG_GET and DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET to get information about the image about to be transferred.
- The application initiates the transfer using a DG_CONTROL / DAT_IMAGExxxFER / MSG_GET operation. The transfer occurs.
- Following a successful transfer, the Source returns TWRC_XFERDONE.
- The application sends the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation to acknowledge the end of the transfer and learn the number of pending transfers.

If the intent behind transferring a single image is to simply flush it from the Source (for example, an application may want to scan only every other page from a stack placed in a scanner with a document feeder), the following operation suffices:

- Issue a CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation. As with normal image transfer, this operation tells the Source that the application has completed acquisition of the current image, and the Source responds by reporting the number of pending transfers.

Preparing for Multiple Image Transfer

The DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation issued by the application at the end of every image transfer performs two important functions:

- It returns a count of pending transfers (in TW_PENDINGXFERS.Count)
- It transitions the session to State 6 (Transfer Ready) if the count of pending transfers is nonzero, or to State 5 (Source Enabled) if the count is zero. Recall that the count returned is a positive value if the Source knows the number of images available for acquisition. If the Source does not know the number of images available, the count returned us -1. The latter situation can occur if, for example, a document feeder is in use. Note that not knowing the number of images available includes the possibility that no further images are available; see the description of DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER for more on this.

We have just seen that after the MSG_ENDXFER operation is issued following an image transfer, the session is either in State 6 or State 5; that is, the session is still very much in an active state. If the session is in State 6 (i.e. “an image is available”), the application takes one of two actions so as to eventually transition the session to State 5 (i.e. “Source is ready to acquire an image, though none is available”):

- It continues to perform the single-image transfer process outlined earlier until no more images are available, or
- It issues a DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET to flush all pending transfers from the Source.

Once the session is back in State 5, the application has to decide whether to stay in State 5 or transition down to State 4 ("Source is open, and ready for capability negotiation".) Two scenarios are possible here.

In one scenario, the application lets the Source control further state transitions. If the Source sends it a MSG_XFERREADY, the application restarts the multiple image transfer loop described above. If the Source sends it a MSG_CLOSEDSREQ (e.g. because the user activated the "Done" trigger on the UI displayed by the Source), the application sends back a DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS, thereby putting the session in State 4.

In the other scenario, the application directly controls session state transitions. For example, the application may want to shut down the current session as soon as the current batch of images have been transferred. In this case, the application issues a DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS as soon as the pending transfers count reaches zero.

It should be noted that there is no "right", "wrong" or "preferred" scenario for an application to follow when deciding what to do once all images in the current set have been transferred. If an application wants to let the user control the termination of a session explicitly, it may well wait for the Source to send it a MSG_CLOSEDSREQ. On the other hand, the application may have a strong sense of what constitutes a session; for example, it may want to terminate a scan session as soon as a blank page is transferred. In such a case, the application will want to control the condition under which the MSG_DISABLEDS is sent.

Use of a Document Feeder

The term document feeder can refer to a physical device's automatic document feeder, such as might be available with a scanner, or to the logical feeding ability of an image database. Both input mechanisms apply although the following text uses the physical feeder for its discussion. The topics covered in this section are:

- Controlling whether to scan pages from the document feeder or the platen
- Detecting whether or not paper is ready for scanning
- Controlling scan lookahead

Note that these concepts are applicable to scanners that do not have feeders; see the discussion below for details.

Selecting the Document Feeder

Sometimes the use of a document feeder actually alters how the image is acquired. For instance, a scanner may move its light bar over a piece of paper if the paper is placed on a platen. When a document feeder is used, however, the same scanner might hold the light bar stable and scan the moving paper. To prepare for such variations the application and Source can explicitly agree to use the document feeder. The negotiation for this action must occur during State 4 **before** the Source is enabled using the following capability.

CAP_FEEDERENABLED

Determine if a Source has a document feeder available and, if so, select that option.

- To determine if this capability is supported, use a DG_CONTROL / DAT_CAPABILITY / MSG_GET operation. TWRC_FAILURE /

TWCC_BADCAP indicates this Source does not have the ability to select the document feeder.

- If supported, use the DG_CONTROL / DAT_CAPABILITY / MSG_SET operation during State 4.
- Set TW_CAPABILITY.Cap to CAP_FEEDERENABLED.
- Create a container of type TW_ONEVALUE and set it to TRUE. Reference TW_CAPABILITY.hContainer to the container.
- Execute the MSG_SET operation and check the Return Code.

  If TWRC_SUCCESS then the feeder is available and your request to use it was accepted. The application can now set other document feeder capabilities.

  If TWRC_FAILURE and **TWCC_BADCAP** or **TWCC_BADVALUE** then this Source does not have a document feeder capability or does not allow it to be selected explicitly.

Note: If an application wanted to prevent the user from using a feeder, the application should use a MSG_SET operation to set the CAP_FEEDERENABLED capability to FALSE.

Detecting Whether an Image is Ready for Acquisition

Having an image ready for acquisition in the Source device is independent of having a selectable document feeder. There are three possibilities here:

- The Source cannot tell whether an image is available,
- An image is available for acquisition, or
- No image is available for acquisition

These cases can be detected by first determining whether a Source can tell that image data is available for acquisition (case 1. vs. cases 2. and 3.) and then determining whether image data is available (case 2. vs. case 3.)The capabilities used to do so are as follows:

CAP_PAPERDETECTABLE

First, determine if the Source can tell that documents are loaded.

- To check if a Source can detect documents, use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation.
- Set the TW_CAPABILITY.Cap field to CAP_PAPERDETECTABLE.
- The Source returns **TWRC_SUCCESS** with the hContainer structure's value set to TRUE if it can detect a loaded document that is ready for acquisition. If the result code is **TWRC_FAILURE** with **TWCC_BADCAP** or **TWCC_BADVALUE**, then the Source cannot detect that paper is loaded.

Note: CAP_PAPERDETECTABLE can be used independently of CAP_FEEDERENABLED. Also, an automatic document feeder need not be present for a Source to support this capability; e.g. a scanner that can detect paper on its platen should return TRUE.

The application cannot set this capability. The Source is simply reporting on a condition.

CAP_FEEDERLOADED

Next, determine if there are documents loaded in the feeder.

- To check if pages are present, use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation.
- Set the TW_CAPABILITY.Cap field to CAP_FEEDERLOADED.
- The Source returns TRUE if there are documents loaded. The information is in the container structure pointed to by the hContainer field of the TW_CAPABILITY structure.

Note: Neither CAP_FEEDERENABLED nor CAP_PAPERDETECTABLE need be TRUE to use this capability. A FALSE indication from this capability simply indicates that the feeder is not loaded or that the Source's feeder cannot tell. For a definitive answer, be sure to check CAP_PAPERDETECTABLE.

Controlling Scan Lookahead

With low-end scanners there is usually ample time for the CPU handling the image acquisition to process incoming image data on-the-fly or in the scan delay between pages. However, with higher performance scanners the CPU image processing time for a given page can become a significant fraction of the scan time. This problem can be alleviated if the scanner can scan ahead image data that the CPU has yet to acquire. This data can be buffered in scanner-local memory, or stored in main memory by the Source via a DMA operation while the CPU processes the current image.

Scan look-ahead is not always desirable, however. This is because the decision to continue a scan may be determined by the results of previously scanned images. For example, a scanning application may decide to stop a scan whenever it sees a blank page. If scan look-ahead were always enabled, one or more pages past the blank page may be scanned and transferred to the scanner's output bin. Such behavior may be incorrect from the point of view of the application's design

We have argued that the ability to control scan look-ahead is highly desirable. However, a single "enable scan look-ahead" command is insufficient to capture the richness of function provided by some scanners. In particular, TWAIN's model of document feeding has each image (e.g., sheet of paper) transition through a three stage process.

1. *Image is in input bin.* This action is taken by the user (for example, by placing a stack of paper into an auto-feeder.)
2. *Image is ready for scan.* This action causes the next available image to be placed at the start of the scan area. Set the CAP_AUTOFEED capability(described below)to automatically feed images to the start of the scan area.
3. *Image is scanned.* This action actually causes the image to be scanned. For example, the DG_IMAGE/DAT_IMAGEMEMXFER/MSG_GET operation initiates image transfer to an application via buffered memory. TWAIN allows a Source to pre-fetch images into Source-local memory (even before the application requests them) by setting the CAP_AUTOSCAN capability.

CAP_AUTOFEED

Enable the Source's automatic document feeding process.

- Use DG_CONTROL / DAT_CAPABILITY / MSG_SET.
- Set the TW_CAPABILITY.Cap field to CAP_AUTOFEED and set the capability to TRUE.
- When set to TRUE, the behavior of the Source is to eject one page and feed the next page after all frames on the first page are acquired. This automatic feeding process will continue whenever there is image data ready for acquisition (and the Source is in an enabled state). CAP_FEEDERLOADED is TRUE showing that pages are in the document feeder.

Note: CAP_FEEDERENABLED must be set to TRUE to use this capability. If not, the Source should return TWRC_FAILURE / TWCC_BADCAP.

CAP_AUTOSCAN

Enable the Source's automatic document scanning process.

- Use DG_CONTROL / DAT_CAPABILITY / MSG_SET.
- Set the TW_CAPABILITY.Cap field to CAP_AUTOSCAN and set the capability to TRUE.
- When set to TRUE, the behavior of the Source is to eject one page and scan the next page after all frames on the first page are acquired. This automatic scanning process will continue whenever there is image data ready for acquisition (and the Source is in an enabled state.

Note: Setting CAP_AUTOSCAN to TRUE implicitly sets CAP_AUTOFEED to TRUE also.

When your application uses automatic document feeding:

- Set CAP_XFERCOUNT to -1 indicating your application can accept multiple images.
- Expect the Source to return the TW_PENDINGXFERS.Count as -1. It indicates the Source has more images to transfer but it is not sure how many.
- Using automatic document feeding does not change the process of transferring multiple documents described earlier and in Chapter 3.

Control of the Document Feeding by the Application

In addition to automatic document feeding, TWAIN provides an option for an application to manually control the feeding of documents. This is only possible if the Source agrees to negotiate the following capabilities during States 5 and 6 by use of CAP_EXTENDEDCAPS. If CAP_AUTOFEED is set to TRUE, it can impact the way the Source responds to the following capabilities as indicated below.

CAP_FEEDPAGE

- If the application sets this capability to TRUE, the Source will eject the current page (if any) and feed the next page.
- To work as described requires that CAP_FEEDERENABLED and CAP_FEEDERLOADED be TRUE.
- If CAP_AUTOFEED is TRUE, the action is the still the same.
- The page ejected corresponds to the image that the application is acquiring (or is about to acquire). Therefore, if CAP_AUTOSCAN is TRUE and one or more pages have been scanned speculatively, the page ejected may correspond to a page that has already been scanned into Source-local buffers.

CAP_CLEARPAGE

- If the application sets this capability to TRUE, the Source will eject the current page and leave the feeder acquire area empty (that is, with no image ready to acquire).
- To work as described, this requires that CAP_FEEDERENABLED be TRUE and there be a paper in the feeder acquire area to begin with.
- If CAP_AUTOFEED is TRUE, the next page will advance to the acquire area.
- If CAP_AUTOSCAN is TRUE, setting this capability returns **TWRC_FAILURE** with **TWCC_BADVALUE**.

CAP_REWINDPAGE

- If the application sets this capability to TRUE, the Source will return the current page to the input area and return the last page from the output area into the acquisition area.
- To work as described requires that CAP_FEEDERENABLED be TRUE.
- If CAP_AUTOFEED is TRUE, the normal automatic feeding will continue after all frames of this page are acquired.
- The page rewound corresponds to the image that the application is acquiring. Therefore, if CAP_AUTOSCAN is TRUE and one or more pages have been scanned speculatively, the page rewound may correspond to a page that has already be scanned into Source-local buffers.

Transfer of Compressed Data

When using the Buffered Memory mode for transferring images, some Sources may support the transfer of data in a compressed format.

To determine if a Source supports transfer of compressed data and to set the capability

1. Use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation.
2. Set the TW_CAPABILITY.Cap field to ICAP_COMPRESSION.
3. The Source returns information about the compression schemes they support in the container structure pointed to by the hContainer field of TW_CAPABILITY. The identifiers for the compression alternatives all begin with TWCP_, such as TWCP_PACKBITS, and can be seen in the Constants section of Chapter 8 and in the TWAIN.H file.
4. If you wish to negotiate for the transfer to use one of the compression schemes shown, use the DG_CONTROL / DAT_CAPABILITY / MSG_SET operation.

The TW_IMAGEMEMXFER structure is used with the DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET operation. The structure looks like this:

```
typedef struct {
  TW_UINT16   Compression; /* A TWCP_xxx constant */
  TW_UINT32   BytesPerRow;
  TW_UINT32   Columns;
  TW_UINT32   Rows;
  TW_UINT32   XOffset;
  TW_UINT32   YOffset;
  TW_UINT32   BytesWritten;
  TW_MEMORY   Memory;
} TW_IMAGEMEMXFER, FAR *pTW_IMAGEMEMXFER;
```

When compressed strips of data are transferred:

- The BytesPerRow field will be set to 0. The Columns, Rows, XOffset, and YOffset fields will contain TWON_DONTCARE32 indicating the fields hold invalid values. (The original image height and width are available by using the DG_IMAGE / DAT_IMAGEINFO / MSG_GET operation during State 6 prior to the transfer.)
- Transfer buffers are always completely filled by the Source. For compressed data, it is very likely that at least one partial line will be written into the buffer.
- The application is responsible for deallocating the buffers.

When compressed, tiled data are transferred:

- All fields in the structure contain valid data. BytesPerRow, Columns, Rows, XOffset, and YOffset all describe the uncompressed tile. Compression and BytesWritten describe the compressed tile.
- In this case, unlike with compressed, strip data transfer, the Source allocates the transfer buffers. This allows the Source to create buffers of differing sizes so that complete, compressed tiles can be transferred to the application intact (not split between sequential buffers). Under these conditions, the application should set the fields of the TW_MEMORY structure so Flags is TWMF_DSOWNS, Length is TWON_DONTCARE32 and TheMem is NULL. The Source must assume that

the application will keep the previous buffer rather than releasing it. Therefore, the Source must allocate a new buffer for each transfer.

- The application is responsible for deallocating the buffers.
- Finally, the application cannot assume that the tiles will be transferred in any particular, logical order.

JPEG Compression

TWAIN supports transfer of several forms of compressed data. JPEG compression is one of them. The JPEG compression algorithm provides compression ratios in the range of 10:1 to 25:1 for grayscale and full-color images, often without causing visible loss of image quality. This compression, which is created by the application of a series of "perceptual" filters, is achieved in three stages:

Color Space Transformation and Component Subsampling (Color Images Only, Not for Grayscale)

The human eye is far more sensitive to light intensity (luminance) than it is to light frequency (chrominance, or "color") since it has, on average, 100 million detectors for brightness (the "rods") but only about 6 million detectors for color (the "cones"). Substantial image compression can be achieved simply by converting a color image into a more efficient luminance/chrominance color space and then subsampling the chrominance components.

This conversion is provided for by the TW_JPEGCOMPRESSION structure. By specifying the TW_JPEGCOMPRESSION.ColorSpace = TWPT_YUV, Source RGB data is converted into more space-efficient YUV data (better known as CCIR 601-1 or YCbCr). TW_JPEGCOMPRESSION.SubSampling specifies the ratio of luminance to chrominance samples in the resulting YUV data stream, and a typical choice calls for two luminance samples for every chrominance sample. This type of subsampling is specified by entering 0x21102110 into the TW_JPEGCOMPRESSION.SubSampling field. A larger ratio of four luminance samples for every chrominance sample is represented by 0x41104110. To sample two luminance values for every chrominance sample in both the horizontal and vertical axes, use a value of 0x21102110.

Application of the Discrete Cosine Transform (DCT) and Quantization

The original components (with or without color space conversion) are next mathematically converted into a spatial frequency representation using the DCT and then filtered with quantization matrices (each frequency component is divided by its corresponding member in a quantization matrix). The quantization matrices are specified by TW_JPEGCOMPRESSION.QuantTable[] and up to four quantization matrices may be defined for up to four different original components. TW_JPEGCOMPRESSION.QuantMap[] maps the particular original component to its respective quantization matrix.

Note: Defaults are provided for the quantization map and tables are suggested in Section K of the JPEG Draft International Standard, version 10918-1 are used as the default tables for QuantTable, HuffmanDC, and HuffmanAC by TWAIN. The default tables are selected by putting NULL into each of the TW_JPEGCOMPRESSION.QuantTable[] entries.

Huffman encoding

The resulting coefficients from the DCT and quantization steps are further compressed in one final stage using a loss-less compression algorithm called Huffman encoding. Application developers can provide Huffman tables, though typically the default tables—selected by writing NULL into TW_JPEGCOMPRESSION.HuffmanDC[] and TW_JPEGCOMPRESSION.HuffmanAC[]—yield very good results.

The algorithm optionally supports the use of restart marker codes. The purpose of these markers is to allow random access to strips of compressed data in JPEG data stream. They are more fully described in the JPEG specification.

See Chapter 8 for the definition of the TW_JPEGCOMPRESSION data structure. Example data structures are shown below for RGB image compression and grayscale image compression:

```
/* RGB image compression - YUV conversion and 2:1:1 chrominance */
/* subsampling                                                  */

typedef struct TW_JPEGCOMPRESSION  myJPEG;

myJPEG.ColorSpace       = TWPT_YUV;        // convert RGB to YUV
myJPEG.SubSampling      = 0x21102110;      // 2 Y for each U, V
myJPEG.NumComponents    = 3;               // Y, U, V
myJPEG.RestartFrequency = 0;               // No restart markers
myJPEG.QuantMap[0]      = 0;               // Y component uses table0
myJPEG.QuantMap[1]      = 1;               // U component uses table 1
myJPEG.QuantMap[2]      = 1;               // V component uses table 1
myJPEG.QuantTable[0]    = NULL;            // select defaults for quant
                                           // tables
myJPEG.QuantTable[1]    = NULL;            //
myJPEG.QuantTable[2]    = NULL;            //
myJPEG.HuffmanMap[0]    = 0;               // Y component uses DC & AC
                                           // table 0
myJPEG.HuffmanMap[1]    = 1;               // U component uses DC & AC
                                           // table 1
myJPEG.HuffmanMap[2]    = 1;               // V component uses DC & AC
                                           // table 1
myJPEG.HuffmanDC[0]     = NULL;            // select default for Huffman
                                           // tables
myJPEG.HuffmanDC[1]     = NULL;            //
myJPEG.HuffmanAC[0]     = NULL;            //
myJPEG.HuffmanAC[1]     = NULL;            //

/* Grayscale image compression - no color space conversion or  */
/* subsampling                                                 */

typedef struct TW_JPEGCOMPRESSION  myJPEG;

myJPEG.ColorSpace       = TWPT_GRAY;       // Grayscale data
myJPEG.SubSampling      = 0x10001000;      // no chrominance components
myJPEG.NumComponents    = 1;               // Grayscale
myJPEG.RestartFrequency = 0;               // No restart markers
myJPEG.QuantMap[0]      = 0;               // select default for quant
                                           // map
myJPEG.QuantTable[0]    = NULL;            //
myJPEG.HuffmanMap[0]    = 0;               // select default for Huffman
                                           // tables
myJPEG.HuffmanDC[0]     = NULL;            //
myJPEG.HuffmanAC[0]     = NULL;            //
```

The resulting compressed images from these examples will be compatible with the JPEG File Interchange Format (JFIF version 1.1) and will therefore be usable by a variety of applications that are JFIF-aware.

Alternative User Interfaces

Alternatives to Using the Source Manager's Select Source Dialog

TWAIN ships its Source Manager code to act as the communication vehicle between application and Source. One of the services the Source Manager provides is locating all available Sources that meet the application's requirements and presenting those to the user for selection.

It is recommended that the application use this approach. However, the application is not required to use this service. Two alternatives exist:

- The application can develop and present its own custom selection interface to the user. This is presented in response to the user choosing **Select Source...** from its menu.
- Or, if the application is dedicated to control of a specific Source, the application can transparently select the Source. In this case, the application does not functionally need to have a **Select Source...** option in the menu but a grayed-out one should be displayed for consistency with all other TWAIN-compliant applications.

Displaying a custom selection interface:

1. Use the DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST operation to have the Source Manager locate the first Source available. The name of the Source is contained in the TW_IDENTITY.ProductName field. Save the TW_IDENTITY structure.
2. Use the DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT to have the Source Manager locate the next Source. Repeatedly use this operation until it returns TWRC_ENDOFLIST indicating no more Sources are available. Save the TW_IDENTITY structure.
3. Use the ProductName information to display the choices to the user. Once they have made their selection, use the saved TW_IDENTITY structure and the DG_CONTROL / DAT_IDENTITY / MSG_OPENDS operation to have the Source Manager open the desired Source. (Note, using this approach, as opposed to the MSG_USERSELECT operation, the Source Manager does not update the system default Source information to reflect your choice.)

Transparently selecting a Source:

If the application wants to open the system default Source , use the DG_CONTROL / DAT_IDENTITY / MSG_GETDEFAULT operation to have the Source Manager locate the default Source and fill the TW_IDENTITY structure with information about it. The name of the Source is contained in the TW_IDENTITY.ProductName field. Save the TW_IDENTITY structure.

OR - If you know the ProductName of the Source you wish to use and it is not the system default Source, use the DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST and DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT operations to have the

Source Manager locate each Source. You must continue looking at Sources until you verify that the desired Source is available. Save the TW_IDENTITY structure when you locate the Source you want. If the Return Code TWRC_ENDOFLIST appears before the desired Source is located, it is not available.

Use the saved TW_IDENTITY structure and the DG_CONTROL / DAT_IDENTITY / MSG_OPENDS operation to have the Source Manager open the desired Source. (Note, using this approach, rather than MSG_USERSELECT, the Source Manager does not update the system default Source information to reflect your choice.)

Alternatives to Using the Source's User Interface

Just as with the Source Manager's Select Source dialog, the application may ask to not use the Source's user interface. Certain types of applications may not want to have the Source's user interface displayed. An example of this can be seen in some text recognition packages that wish to negotiate a few capabilities (i.e. pixel type, resolution, page size) and then proceed directly to acquiring and transferring the data.

To Enable the Source without Displaying its User Interface

- Use the DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS operation.
- Set the ShowUI field of the TW_USERINTERFACE structure to FALSE.
- When the command is received and accepted (TWRC_SUCCESS), the Source does not display a user interface but is armed to begin capturing data. For example, in a flatbed scanner, the light bar will light and begin to move. A handheld scanner will be armed and ready to acquire data when the "go" button is pressed on the scanner. Other devices may respond differently but they all will either begin acquisition immediately or be armed to begin acquiring data as soon as the user interacts with the device.

Capability Negotiation is Essential when the Source's User Interface is not Displayed

- Since the Source's user interface is not displayed, the Source will not be giving the user the opportunity to select the information to be acquired, etc. Unless default values are acceptable, current values for all image acquisition and control parameters must be negotiated before the Source is enabled, i.e. while the session is in State 4.

When TW_USERINTERFACE.ShowUI is set to FALSE:

- The application is still required to pass all events to the Source (via the DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation) while the Source is enabled.
- The Source must display the minimum possible user interface containing only those controls required to make the device useful in context. In general, this means that no user interface is displayed, however certain devices may still require a trigger to initiate the scan.
- The Source still displays a progress indicator during the acquisition. The application can suppress this by setting CAP_INDICATORS to FALSE, if the Source allows this.
- The Source still displays errors and other messages related to the operation of its device. This cannot be turned off.
- The Source still sends the application a MSG_XFERREADY notice when the data is ready to be transferred.
- The Source may or may not send a MSG_CLOSEDSREQ to the application asking to be closed since this is often user-initiated. Therefore, after the Source has returned to State 5 (following the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation and the TW_PENDINGXFERS.Count = 0), the application can send the DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS operation.

Note: Some Sources may display the UI even when ShowUI is set to FALSE. An application can determine whether ShowUI can be set by interrogating the CAP_UICONTROLLABLE capability. If CAP_UICONTROLLABLE returns FALSE but the ShowUI input value is set to FALSE in an activation of DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS, the enable DS operation returns TWRC_CHECKSTATUS but displays the UI regardless. Therefore, an application that requires that the UI be disabled should interrogate CAP_UICONTROLLABLE before issuing MSG_ENABLEDS.

Modal Versus Modeless User Interfaces

The Source Manager's user interface is a modal interface but the Source may provide a modeless or modal interface. Here are the differences:

Modeless

When a Source uses a modeless user interface, although the Source's interface is displayed, the user is still able to access the application by clicking on the application's window and making it active.

The user is expected to click on a Close button on the Source's user interface when they are ready for that display to go away. The application must NOT automatically close a modeless Source after the first (or any subsequent) transfer, even if the application is only interested in receiving a single transfer. If the application closes the Source before the user requests it, the user is likely to become confused about why the window disappeared. Wait until the user indicates the desire to close the Source's window and the Source sends this request (MSG_CLOSEDSREQ) to the application before closing the Source.

Modal

A Source using a modal user interface prevents the user from accessing other windows.

For Windows only, if the interface is application modal, the user cannot access other applications but can still access system utilities. If the interface is system modal (which is rare), the user cannot access anything else at an application or system level. A system modal dialog might be used to display a serious error message, like a UAE.

If using a modal interface, the Source can perform only one acquire during a session although there may be multiple frames per acquisition. The Source will send a close request to the application following the completion of the data transfer. Again, the application waits to receive this request.

The Source indicates if it is using a modeless or modal interface after the application enables it using the DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS operation. The data structure used in the operation (TW_USERINTERFACE) contains a field, ShowUI, which is set by the application to indicate whether the Source should display its user interface. If the application requests the user interface be shown, the Source sets the ModalUI field to indicate if its user interface is modal (TRUE) or modeless (FALSE).

When requested by the Source, the application uses the DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS operation to remove the Source's user interface.

Grayscale and Color Information for an Image

There are operation triplets in TWAIN that allow the application developer to interact with and influence the grayscale or color aspect of the images that a Source transfers to the application. The following operations provide these abilities:

CIE Color Descriptors

DG_IMAGE / DAT_CIECOLOR / MSG_GET

Grayscale Changes

DG_IMAGE / DAT_GRAYRESPONSE / MSG_RESET
DG_IMAGE / DAT_GRAYRESPONSE / MSG_SET

Palette Color Data

DG_IMAGE / DAT_PALETTE8 / MSG_GET
DG_IMAGE / DAT_PALETTE8 / MSG_GETDEFAULT
DG_IMAGE / DAT_PALETTE8 / MSG_RESET
DG_IMAGE / DAT_PALETTE8 / MSG_SET

RGB Response Curve Data

DG_IMAGE / DAT_RGBRESPONSE / MSG_RESET
DG_IMAGE / DAT_RGBRESPONSE / MSG_RESET

CIE Color Descriptors

The CIE XYZ approach is a method for storing color data which simplifies doing mathematical manipulations on the data. (The topic of CIE XYZ color space is discussed thoroughly in Appendix A.)

If your application wishes to receive the image data in this format:

1. You must ensure that the Source is able to provide data in CIE XYZ format. To check this, use the DG_CONTROL / DAT_CAPABILITY / MSG_GET operation and get information on the ICAP_PIXELTYPE. If TWPT_CIEXYZ is returned as one of the supported types, the Source can provide data in CIE XYZ format.
2. After verifying that the Source supports it, the application can specify that data transfers should use the CIE XYZ format by invoking a DG_CONTROL / DAT_CAPABILITY / MSG_SET operation on the ICAP_PIXELTYPE. Use a TW_ONEVALUE container whose value is TWPT_CIEXYZ.

To determine the parameters that were used by the Source in converting the color data into the CIE XYZ format, use the DG_IMAGE / DAT_CIECOLOR / MSG_GET operation following the transfer of the image.

Grayscale Changes

(The grayscale operations assume that the application has instructed the Source to provide grayscale data by setting its ICAP_PIXELTYPE to TWPT_GRAY and the Source is capable of this.)

The application can request that the Source apply a transfer curve to its grayscale data prior to transferring the data to the application. To do this, the application uses the DG_IMAGE / DAT_GRAYRESPONSE / MSG_SET operation. The desired transfer curve information is placed by the application within the TW_GRAYRESPONSE structure (the actual array is of type TW_ELEMENT8). The application must be certain to check the Return Code following this request. If the Return Code is TWRC_FAILURE and the Condition Code shows TWCC_BADPROTOCOL, this indicates the Source does not support grayscale response curves (despite supporting grayscale data).

If the Source allows the application to set the grayscale transfer curve, there must be a way to reset the curve to its original non-altered value. Therefore, the Source must have an "identity response curve" which does not alter grayscale data but transfers it exactly as acquired. When the application sends the DG_IMAGE / DAT_GRAYRESPONSE / MSG_RESET operation, the Source resets the grayscale response curve to its identity response curve.

Palette Color Data

(The palette8 operations assume that the application has instructed the Source to use the TWPT_PALETTE type for its ICAP_PIXELTYPE and that the Source has accepted this.)

The DAT_PALETTE8 operations allow the application to inquire about a Source's support for palette color data and to set up a palette color transfer. The operations are specialized for 8-bit data, whether grayscale or color (8-bit or 24-bit). The MSG_GET operation allows the application to learn what palette was used by the Source during the image acquisition. The application should always execute this operation immediately after an image transfer rather than before because the Source may optimize the palette during the acquisition process. Some Sources may allow an application to define the palette to be used during image acquisition via the MSG_SET operation. Be sure to check the Return Code to verify that it is TWRC_SUCCESS following a MSG_SET operation. That is the only way to be certain that your requested palette will actually be used during subsequent palette transfers.

RGB Response Curve Data

(The RGB Response curve operations assume that the application has instructed the Source to provide RGB data by setting its ICAP_PIXELTYPE to TWPT_RGB and the Source is capable of this.)

The application can request that the Source apply a transfer curve to its RGB data prior to transferring the data to the application. To do this, the application uses the DG_IMAGE / DAT_RGBRESPONSE / MSG_SET operation. The desired transfer curve information is placed by the application within the TW_RGBRESPONSE structure (the actual array is of type TW_ELEMENT8). The application must be certain to check the Return Code following this request. If the Return Code is TWRC_FAILURE and the Condition Code shows TWCC_BADPROTOCOL, this indicates the Source does not support RGB response curves (despite supporting RGB data).

If the Source allows the application to set the RGB response curve, there must be a way to reset the curve to its original non-altered value. Therefore, the Source must have an "identity response curve" which does not alter RGB data but transfers it exactly as acquired. When the application sends the DG_IMAGE / DAT_RGBRESPONSE / MSG_RESET operation, the Source resets the RGB response curve to its identity response curve.

5

Source Implementation

Companies who produce image-acquisition devices, and wish to gain the benefits of being TWAIN-compliant, must develop TWAIN-compliant Source software to drive their device. The Source software is the interface between TWAIN's Source Manager and the company's physical (or logical) device. To write effective Source software, the developer must be familiar with the application's expectations as discussed in the other chapters of this document. This chapter discusses:

Chapter Contents

The Structure of a Source

The following sections describe the structure of a source.

On Windows

Implementation

The Source is implemented as a Dynamic Link Library (DLL). Sources should link to TWAIN.LIB at link time. The Source will not run stand-alone. The DLL typically runs within the (first) calling application's heap although DLLs may be able to allocate their own heap and stack space. There is only one copy of the DLL's code and data loaded at run-time, even if more than one application accesses the Source. For more information regarding DLLs on Win32s, Windows95, and Windows NT please refer to Microsoft documents.

Naming and Location

The DLL's file name must end with a .DS extension. The Source Manager recursively searches for your Source in the TWAIN sub-directory of the Windows directory (or whatever the name of the directory in which WIN.COM resides). To reduce the chance for naming collisions, each Source should create a sub-directory beneath TWAIN, giving it a name relevant to their product. The Source DLLs and supporting files are placed there.

Entry Points and Segment Attributes

Every Source is required to have a single entry point called DS_Entry (see Chapter 6). In order to speed up the Source Manager's ability to identify Sources, the Source entry point DS_Entry( ) and the code to respond to the DG_CONTROL / DAT_IDENTITY / MSG_GET operation must reside in a segment marked as PRELOAD. All other segments should be marked as LOADONCALL (with the exception of any interrupt handler that may exist in the Source which needs to be in a FIXED code segment).

Resources

- **Version Information** - The Microsoft VER.DLL is included with the TWAIN toolkit for use by your installation program, if you have one, to validate the version of the currently installed Source Manager. Sources should be marked with the Version information capability defined in MS Windows 3.1. To do this, you can use the resource compiler from the version 3.1 SDK. VER.DLL and the version stamping are also compatible with MS Windows version 3.0.
- **Icon Id** - Future versions of the TWAIN Source Manager may display the list of available Sources using a combination of the ProductName (in the Source's TW_IDENTITY structure) and an Icon (as the Macintosh version currently does). Therefore, it is recommended that you add this icon into your Source resource file today. This will allow your Source to be immediately compatible with any upcoming changes. The icon should be identified using TWON_ICONID from the TWAIN.H file.

General Notes

- **GlobalNotify -** MS Windows allows only one GlobalNotify handler per task. As the Source resides in the application heap, the Source should **not** use the GMEM_NOTIFY flag on the memory blocks allocated as this may disrupt the correct behavior of the application that uses GlobalNotify.
- **Windows Exit Procedure (WEP) -** During the WEP, the Source is being unloaded by MS Windows. The Source should make sure all the resources it allocated and owns get released whether or not the Source was terminated properly.

On Macintosh

Implementation

A Source on a Macintosh is implemented as a Code Resource. The Source will not run stand-alone. The Source's code will exist and run within the calling application's heap. A separate copy of the Source's code will be made for each application that opens the Source. Macintosh development books such as *Inside Macintosh* describe the special requirements for developing Code Resources.

A Source can communicate between various running copies of itself through its resource file. If a Source must enforce a maximum number of applications that the device it controls can support, for instance, it can maintain the number of current connects within the Source's resource fork. If the maximum number of connects is exceeded when a new application tries to open a new copy of the Source, the Source should display an appropriate error message to the user and then return TWRC_FAILURE with a Condition Code of TWCC_MAXCONNECTIONS. This gives each running copy visibility of the total number of connections. Other information that needs to be communicated between instances of the Source can use this same method.

Naming and Location

The type for a Source is DSRC. The Source Manager will recursively search for files of this type in the TWAIN folder. The creator is vendor-dependent.

Under System 6.x, Sources are located within the TWAIN folder which resides within the System Folder. It is recommended that each Source file, along with any other files it may require, be installed into a uniquely named folder within the TWAIN folder. Placing your files within a specially named folder will limit the chances of name collisions of the Source's support files (or the Source itself) with the names of other Sources and Source-support files already installed. The Source Manager will recursively search out all Sources within the TWAIN folder.

Under System 7, the TWAIN folder is located within the Preferences folder (which resides within the System Folder). All Sources should be located within the Preferences folder per the rules of the preceding paragraph. The Source Manager can sense the version of the operating system and will look in the correct folder.

Entry Point

The entry point for the code resource, DS_Entry, **must** be the first address in the resource and **must** have the format described in Chapter 6.

Resources

- **Version Information -** The Macintosh Source Manager will keep its version information in it **vers** resource. The application can read this resource for Source Manager information. The resource ID of this resource is in the constant TWON_DSMID (see the TWAIN.H file).
- **Code Resources -** The initial code resource that will be loaded by the Source Manager must have a resource type of DSRC. This resource must have a resource ID defined in the constant TWON_DSRCID (see the TWAIN.H file). The first address in this resource must be DS_Entry, or must call it directly.
- **ICN# Resource** - The icon that is displayed by the Source Manager in the Select Source... dialog is an ICN# resource with a resource ID defined in the constant TWON_ICONID (see the TWAIN.H file).

Operation Triplets

In Chapter 3, we introduced all of the triplets that an application can send to the Source Manager or ultimately to a Source. There are several other triplet operations which do not originate from the application. Instead, they originate from the Source Manager or Source and are introduced in this chapter. All defined operation triplets are listed in detail in Chapter 7.

Triplets from the Source Manager to the Source

There are three operation triplets that are originated by the Source Manager. They are:

DG_CONTROL / DAT_IDENTITY

MSG_GET :	Returns the Source's identity structure
MSG_OPENDS :	Opens and initializes the Source
MSG_CLOSEDS :	Closes and unloads the Source

The DG_CONTROL / DAT_IDENTITY / MSG_GET operation is used by the Source Manager to identify available Sources. It may send this operation to the Source **at any time** and the Source **must be prepared** to respond informatively to it. That means, the Source must be able to return its identity structure before being opened by the Source Manager (with the MSG_OPENDS command). The Source's initially loaded code segment must be able to perform this function without loading any additional code segments. This allows quick identification of all available Sources and is the only operation a Source must support before it is formally opened.

The TW_IDENTITY structure looks like this:

```
typedef struct {
      TW_UINT32     Id;
      TW_VERSION     Version;
      TW_UINT16     ProtocolMajor;
      TW_UINT16     ProtocolMinor;
      TW_UINT32     SupportedGroups;
      TW_STR32      Manufacturer;
      TW_STR32      ProductFamily;
      TW_STR32      ProductName;
} TW_IDENTITY, FAR *pTW_IDENTITY;
```

The ProductName field in the Source's TW_IDENTITY structure should uniquely identify the Source. This string will be placed in the Source Manager's Select Source... dialog for the user. (The file name of the Source should also approximate the ProductName, if possible, to add clarity for the user at installation time.) Fill in all fields except the Id field which will be assigned by the Source Manager. The unique Id number that identifies your Source during its current session will be passed to your Source when it is opened by the MSG_OPENDS operation. Sources on Windows must save this TW_IDENTITY.Id information for use when sending notifications from the Source to the application.

Sources and the Event Loop

Handling Events

On both Windows and Macintosh, when a Source is enabled (i.e. States 5, 6, and 7), the application must pass all events (messages) to the Source. Since the Source runs subservient to the application, this ensures that the Source will receive all events for its window. The event will be passed in the TW_EVENT data structure that is referenced by a DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT command.

Routing all messages to all connected Sources while they are enabled places a burden on the application and creates a potential performance bottleneck. Therefore, the Source must process the incoming events as quickly as possible. The Source should examine each incoming operation before doing anything else. Only one operation's message field says MSG_PROCESSEVENT so always look at the message field first. If it indicates MSG_PROCESSEVENT then:

Immediately determine if the event belongs to the Source

If it does

Set the Return Code for the operation to TWRC_DSEVENT

Set the TWMessage field to MSG_NULL

Process the event

Else

Set the Return Code to TWRC_NOTDSEVENT

Set the TWMessage field to MSG_NULL

Return to the application immediately

If the Source developer fails to process events with this high priority, the user may see degraded performance whenever the Source is frontmost which reflects poorly on the Source.

On Windows, the code fragment looks like the following:

```
TW_UINT16 CALLBACK DS_Entry(pTW_IDENTITY pSrc,
                                        TW_UINT32 DG,
                                        TW_UINT16 DAT,
                                        TW_UINT16 MSG,
                                        TW_MEMREF pData)
{
    TWMSG     twMsg;
    TW_UINT16 twRc;

    // Valid states: 5 -- 7.  As soon as the application has enabled the
    // Source it must being sending the Source events.  This allows the
    // Source to receive events to update its user interface and to
    // return messages to the application. The app sends down ALL
    // message, the Source decides which ones apply to it.
if (MSG == MSG_PROCESSEVENT)
    {
          if (hImageDlg && IsDialogMessage(hImageDlg,
(LPMSG)(((pTW_EVENT)pData)->pEvent)))
        {
             twRc = TWRC_DSEVENT;

            // The source should, for proper form, return a MSG_NULL for
            // all Windows messages processed by the Data Source

            ((pTW_EVENT)pData)->TWMessage = MSG_NULL;
        }
        else
        {
                  // notify the application that the source did not
                  // consume this message
                  twRc = TWRC_NOTDSEVENT;
          ((pTW_EVENT)pData)->TWMessage = MSG_NULL;
        }
             }
             else
             {
          // This is a Twain message, process accordingly.
          // The remainder of the Source's code follows...
             }

return twRc;
}
```

The Windows IsDialogMessage( ) call is used in this example. Sources can also use other Windows calls such as TranslateAccelerator( ) and TranslateMDISYSAccel( ).

Communicating to the Application

As explained in Chapter 3, there are two instances where the Source must originate and transmit a notice to the application:

When it has data ready to transfer (MSG_XFERREADY)

The Source must send this message when the user clicks the "GO" button on the Source's user interface or when the application sends a DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS operation with ShowUI = FALSE. The Source will transition from State 5 to State 6. The application should now perform their inquiries regarding TW_IMAGEINFO and capabilities. Then, the application issues a DG_IMAGE / DAT_IMAGExxxXFER / MSG_GET operation to begin the transfer process. Typically, though it is not required, it is at this time that a flatbed scanner (for example) will begin **simultaneously** to acquire and transfer the data using the specified transfer mode.

When it needs to have its user interface disabled (MSG_CLOSEDSREQ)

Typically, the Source will send this only when the user clicks on the "CLOSE" button on the Source's user interface or when an error occurs which is serious enough to require terminating the session with the application. The Source should be in (or transition to) State 5. The application should respond by sending a DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS operation to transition the session back to State 4.

These notices are sent differently on Windows versus Macintosh systems.

On Windows

The Source creates a call to DSM_Entry (the entry point in the Source Manager) and fills the destination with the TW_IDENTITY structure of the application. The Source uses one of the following triplets:

DG_CONTROL / DAT_NULL / MSG_XFERREADY
DG_CONTROL / DAT_NULL / MSG_CLOSEDSREQ

The Source Manager, on Windows, recognizes the notice and makes sure the message is received correctly by the application.

On Macintosh

The Source on Macintosh does not use the operations described above. Instead, it uses a TW_EVENT structure to send its notice to the application. The TW_EVENT structure is created by the application and sent to the Source as data in a DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation.

Normally, the Source places MSG_NULL in the TW_EVENT.TWMessage field. To relay the notice, the Source places one of the following in the TWMessage field:

MSG_XFERREADY
MSG_CLOSEDSREQ

The application examines that field when the DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation concludes and recognizes these two special notices from the Source.

User Interface Guidelines

Each TWAIN-compliant Source provides a user interface to assist the user in acquiring data from their device. Although each device has its own unique needs, the following guidelines are provided to increase consistency among TWAIN-compliant devices.

Displaying the User Interface

The application issues DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS to transition the session from State 4 to 5.

The TW_USERINTERFACE data structure contains these fields:

- **ShowUI** - If set to TRUE, the Source displays its user interface. If FALSE, the application will be providing its own.
- **hParent** - Used by Sources on Windows only. It indicates the application's window handle. This is to be designated as the Source's parent for the dialog so it is a proper child of its parent application.
- **ModalUI** - To be set by the Source to TRUE or FALSE.

Sources are not required to allow themselves to be enabled without showing their user interface (ShowUI = FALSE) but it is strongly recommended that they allow this. If your Source cannot be used without its user interface, it should enable showing the user interface (just as if ShowUI = TRUE) and return TWRC_CHECKSTATUS. All Sources, however, must report whether or not they honor ShowUI set to FALSE via the CAP_UICONTROLLABLE capability. This allows applications to know whether the Source-supplied user interface can be suppressed before it is displayed.

When the user interface is disabled (by DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS), a pointer to a TW_USERINTERFACE is included in the pData parameter.

Modal versus Modeless Interfaces

As stated in Chapter 4, the Source's user interface may be modal or modeless. The modeless approach gives the user more control and is recommended whenever practical. Refer to the information following this table for specifics about Windows and Macintosh implementation.

Error and Device Control Indicators

The Source knows what is happening with the device it controls. Therefore, the Source is responsible for determining when and what information regarding errors and device controls (ex. "place paper in document feeder") should be presented to the user. Error information should be placed by the Source on top of either the application's or Source's user interface, whichever is. Do not present error messages regarding capability negotiation to the user since this should be transparent.

Progress Indicators

The Source should display appropriate progress indicators for the user regarding the acquisition and/or transfer processes. The Source must provide this information regardless of whether or not its user interface is displayed (ShowUI equals TRUE or FALSE). To suppress the indicators when the user interface is not displayed, the application should negotiate the CAP_INDICATORS capability to be FALSE.

Impact of Capability Negotiation

If the Source has agreed to limit the Available Values and/or to set the Current Value, the interface should reflect the negotiation. However, if a capability has not been negotiated by the application, the interface should not be modified (don't gray out a control because it wasn't negotiated.)

Advanced Topics

If a Source can acquire from more than one device, the Source should allow the user to choose which device they wish to acquire from. Provide the user with a selection dialog that is similar to the one presented by the Source Manager's Select Source... dialog.

Implementing Modal and Modeless User Interfaces

On Windows

You cannot use the modal dialog creation call DialogBox( ) to create the Source's user interface main window. To allow event processing by both the application and the Source, this call cannot be used. Modal user interfaces in Source are not inherently bad, however. If a modal user interface makes sense for your Source, use either the CreateDialog( ) or CreateWindow( ) call.

Modal (App Modal)

It is recommended that the Source's main user interface window be created with a modeless mechanism. Source writers can still decide to make their user interface behave modally if they choose. It is even appropriate for a very simple "click and go" interface to be implemented this way.

This is done by first specifying the application's window handle (hWndParent) as the parent window when creating the Source's dialog/window and second by enabling/disabling the parent window during the MSG_ENABLEDS / MSG_DISABLEDS operations. Use EnableWindow(hWndParent, FALSE) to disable the application window and EnableWindow(hWndParent, TRUE) to re-enable it.

Modeless

If implementing a modeless user interface, specify NULL as the parent window handle when creating the Source's dialog/window. Also, it is suggested that you call BringWindowToTop( ) whenever a second request is made by the same application or another application requesting access to a Source that supports multiple application connections.

On Macintosh

It is recommended that the Source's main user interface window be created with a modaless mechanism. Source writers can still decide to make their user interface behave modally if they choose. It is even appropriate for a very simple "click and go" interface to be implemented this way.

Capability Negotiation

Capability negotiation is a critical area for a Source because it allows the application to understand and influence the images that it receives from your Source.

Inquiries From the Application

While the Source is open but not yet enabled (from DG_CONTROL / DAT_IDENTITY / MSG_OPENDS until DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS), the application can inquire all the capabilities and request to set those values.

Once the Source is enabled, the application may only **inquire** about capabilities. An attempt to set a capability should fail with TWRC_FAILURE and TWCC_SEQERROR (unless CAP_EXTENDEDCAPS was negotiated).

Responding to Inquiries

Sources must be able to respond to capability inquiries with current values at any time the Source is open (i.e. from MSG_OPENDS until MSG_CLOSEDS or before posting a MSG_CLOSEDSREQ).

A Source should respond with information to any capability that applies to your device. Only if a capability has no match with your device's features should you respond with TWRC_FAILURE / TWCC_BADCAP.

Refer to Chapter 9 for the complete list of TWAIN-defined capabilities.

Responding to Requests to Set Capabilities

If the requested value is invalid or the Source does not allow the setting (or limiting) of the capability then return TWRC_FAILURE / TWCC_BADVALUE.

If the request was fulfilled, return TWRC_SUCCESS.

If the requested value is close to an acceptable value but doesn't match exactly, set it as closely as possible and then return TWRC_CHECKSTATUS.

Memory Allocation

The TW_CAPABILITY structure used in capability negotiation is both allocated and deallocated by the application. The Container structure pointed to by the hContainer field in TW_CAPABILITY is allocated by the Source for "get" operations (MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, MSG_RESET) and by the application for the MSG_SET operation. Regardless of which one allocates the container, the application is responsible for deallocating it when it is done with it.

Limitations Imposed by the Negotiation

If a Source agrees to allow an application to restrict a capability, it is critical that the Source abide by that agreement. If at all possible, the Source's user interface should reflect the agreement and not offer invalid options. The Source should never transfer data that violates the agreement reached during capability negotiation. In that situation, the Source can decide to fail the transfer or somehow adjust the values.

Data Transfers

Transfer Modes

All Sources must support Native and Buffered Memory data transfers. It is strongly suggested that they support Disk File mode, too. The default mode is Native. To select one of the other modes, the application must negotiate the ICAP_XFERMECH capability (whose default is TWSX_NATIVE). Sources must support negotiation of this capability. The native format for MS Windows is DIB. For Macintosh, the native format is a PICT. The version of PICT to be transferred is the latest version available on the machine on which the application is running (usually PICT II for machines running 32-bit/color QuickDraw and PICT I for machines running black and white QuickDraw).

Initiating a Transfer

Transfers are initiated by the application (using the DG_IMAGE / DAT_IMAGExxxFER / MSG_GET operations). A successful transfer transitions the session to State 7. If the transfer fails, the Source returns TWRC_FAILURE with the appropriate Condition Code and remains in State 6.

Concluding a Successful Transfer

To signal that the transfer is complete (i.e. the file is completed or the last buffer filled), the Source should return TWRC_XFERDONE. In response, the application must send a DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation. Only then may the Source transition from State 7 back to State 6 or to State 5 if no more images are ready to be transferred.

If more images are pending transfer, the Source must wait in State 6 until the application either requests the transfer or aborts the transfers. The Source may not "time-out" on any TWAIN transaction.

Aborting a Transfer

Either the application or Source can originate the termination of a transfer (although the application cannot do this in the middle of a Native or Disk File mode transfer). The Source generally terminates the transfer if the user cancels the transfer or a device error occurs which the Source determines is fatal to the transfer or the connection with the application. If the user canceled the transfer, the Source should return TWRC_CANCEL to signal the premature termination. The session remains in State 7 until the application sends the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER operation. If the Source aborts the transfer, it returns TWRC_FAILURE and the session typically remains in State 6. (If the failure occurs during the second buffer, or a later buffer, of a Buffered Memory transfer, the session remains in State 7.)

Native Mode Transfers

On Native mode transfers, the data parameter in the DSM_Entry call is a pointer to a variable of type TW_UINT32.

On Windows

The low word of this 32-bit integer is a handle variable to a DIB (Device Independent Bitmap) located in memory.

On Macintosh

This 32-bit integer is a handle to a Picture (a PicHandle). It is a Quick Draw picture located in memory.

Native transfers require the data to be transferred to a single large block of RAM. Therefore, they always face the risk of having an inadequate amount of RAM available to perform the transfer successfully.

If inadequate memory prevents the transfer, the Source has these options:

- Fail the transfer operation- Return TWRC_FAILURE / TWCC_LOWMEMORY
- Allow the user to clip the data to fit into available memory - Return TWRC_XFERDONE
- Allow the user to cancel the operation - Return TWRC_CANCEL

If the operation fails, the session remains in State 6. If the operation is canceled, the session remains in State 7 awaiting the DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER or MSG_RESET from the application. The application can return the session to State 4 and attempt to renegotiate the transfer mechanism (ICAP_XFERMECH) to Disk File or Buffered Memory mode.

Disk File Mode Transfers

The Source selects a default file format and file name (typically, TWAIN.TMP in the current directory). It reports this information to the application in response to the DG_CONTROL / DAT_SETUPFILEXFER / MSG_GET operation.

The application may determine all of the Source's supported file formats by using the ICAP_IMAGEFILEFORMAT capability. Based on this information, the application can request a particular file format and define its own choice of file name for the transfer. The desired file format and file name will be communicated to the Source in a DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET operation.

When the Source receives the DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET operation, it should transfer the data into the designated file. The following conditions may exist:

Condition	How to Handle
No file name and/or file format was specified by the application during setup	Use either the Source's default file name or the last file information given to the Source by the application. Create the file.
The application specified a file but failed to create it	Create the application's defined file.
The application's specified file exists but has data in it	Overwrite the existing data.

The Source cannot be interrupted by the application when it is acquiring a file. The Source's user interface may allow the user to abort the transfer, but the application will not be able to do so even if the application presents its own acquisition user interface.

Buffered Memory Mode Transfers

When the Source transfers strips of data, the application allocates and deallocates buffers used for a Buffered Memory mode transfer. However, the Source must recommend appropriate sizes for those buffers and should check that the application has followed its recommendations.

When the Source transfers tiles of data, the Source allocates the buffers. The application is responsible for deallocating the memory.

To determine the Source's recommendations for buffer sizes, the application performs a DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET operation. Fill in the MinBufSize, MaxBufSize, and Preferred fields to communicate your buffer recommendations. Buffers must be double-word aligned and padded with zeros per raster line.

When an application issues a DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET operation, check the TW_IMAGEMEMXFER.Memory.Length field to determine the size of the buffer being presented to you. If it does not fit the recommendations, fail the operation with TWRC_FAILURE / TWCC_BADVALUE.

If the buffer is an appropriate size, fill in the required information.

- Sources must write one or more complete lines of image data (the full width of a strip or tile) into the buffer. Partial lines of image data are not allowed. If some of the buffer is unused, fill it in with zeros. Additionally, each line must be aligned to a 32-bit boundary. Return TWRC_SUCCESS after each successful buffer except for the last one (return TWRC_XFERDONE after that one).
- If the Source is transferring data whose bit depth is not 8 bits, it should fill the buffer without padding the data. If a 5-bit device wants the application to interpret its data as 8-bit data, it should report that it is supplying 8-bit data, make the valid data bits the most significant bits in the data byte, and pad the least significant bits with bits of whichever sense is "lightest". Otherwise, the Source should pack the data bits together. For a 5-bit R-G-B device, that means the data for the green channel should immediately follow the last bit of the red channel. The application is responsible for "unpacking" the data. The Source reports how many bits it is providing per pixel in the BitsPerPixel field of the TW_IMAGEINFO data structure.

Error Handling

Operation Triplet and State Verification

- Sources support all defined TWAIN triplets. A Source must verify every operation triplet they receive. If the operation is not recognized, the Source should return TWRC_FAILURE and TWCC_BADPROTOCOL.
- Sources must also maintain an awareness of what state their session is in. If an application invokes an operation that is invalid in the current state, the Source should fail the operation and return TWRC_FAILURE and TWCC_SEQERROR. Valid states for each operation are listed in Chapter 7.
- Anytime a Source fails an operation that would normally cause the session to transition to another state, the session should not transition but should remain in the original state.
- Each triplet operation has its own set of valid Return and Condition Codes as listed in Chapter 7. The Source must return a valid Return Code and set a valid Condition Code, if applicable, following every operation.
- All Return Codes and Condition Codes in the Source should be cleared upon the next call to DS_Entry( ). Clearing is delayed when a DG_CONTROL / DAT_STATUS / MSG_GET operation is received. In this case, the Source will fill the TW_STATUS structure with the current condition information and then clear that information.
- If an application attempts to connect to a Source that only supports single connection (or a multiply-connected Source that can't establish any new connections), the Source should respond with TWRC_FAILURE and TWCC_MAXCONNECTIONS.
- For Windows Sources only, the DLL implementation makes it possible to be connected to more than one application. Unless the operation request is to open the Source, the Source must verify the application originating an operation is currently connected to the Source. To do this:
  - ✓ The Source must maintain a list containing the Id value for each connected application. (The Id value comes from the application's TW_IDENTITY structure which is referenced by the pOrigin parameter in the DS_Entry( ) call.)
  - ✓ The Source should check the TW_IDENTITY.Id information of the application sending the operation and verify that it appears in the Source's list of connected applications.
- For Windows only, if connected to multiple applications, the Source is responsible for maintaining a separate, current Condition Code for each application it is connected to. The Source writer should also maintain a temporary, and separate, Condition Code for any application that is attempting to establish a connection with the Source. This is true both for Sources that support only a single connection or have reached the maximum connections.

Unrecoverable Error Situations

The Source is solely responsible for determining whether an error condition within the Source is recoverable or not. The Source must determine when, and what, error condition information to present to the user. The application relies on the Source to specify when a failure occurs. If a Source is in an unrecoverable error situation, it may send a MSG_CLOSEDSREQ to the application to request to have its user interface disabled and have an opportunity to begin again.

Memory Management

Windows Specifics

A single copy of the Source Manager and Source(s) services all applications wishing to access TWAIN functionality. If the Source can connect to more than one application, it will probably need to maintain a separate execution frame for each connected application. The Source does not have unlimited memory available to it so be conservative in its use.

It is valid for an application to open a Source and leave it open between several acquires. Therefore, Sources should minimize the time and resources required to load and remain open (in State 4). Also, Sources should allow a reasonable number of connections to ensure they can handle more than one application using the Source in this manner (leaving it open between acquires).

Macintosh Specifics

Each application that loads the Source Manager has a private copy of the Source Manager running within that application's heap space. Each Source that is connected runs as a private copy within the application's heap. It is important for the Source writer to recognize that their Source will be running in the memory frame of the host application, not in its own frame. Therefore, the Source should be conscientious with allocation and deallocation of memory.

General Guidelines

The following are some general guidelines:

- Check, when the Source is launched, to assure that enough memory space is available for adequate execution.
- Always verify that allocations were successful.
- Work with relocatable objects whenever possible - the heap you fragment is not your own.
- Deallocate temporary memory objects as soon as they are no longer needed.
- Maintain as small a non-operating memory footprint as can prudently be done - the Source will be "compatible" with more applications on more machines.
- Clean up after yourself. When about to be closed, deallocate all locally allocated RAM, eliminate any other objects on the heap, and prepare as appropriate to terminate.

Local Variables

The Source may allocate and maintain local variables and buffers. Remember that you are borrowing RAM from the application so be efficient about how much RAM is allocated simultaneously.

Instances Where the Source Allocates Memory

In general, the application allocates all necessary structures and passes them to the Source. There are a few exceptions to this rule:

- The Source must create the container, pointed to by the hContainer field, needed to hold capability information on DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, or MSG_RESET operations. The application deallocates the container.
- The Source allocates the buffer for Native mode data transfers. The application deallocates the buffer.
- Normally, the application creates the buffers used in a Buffered Memory transfer (DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET). However, if the Source is transferring tiled data, rather than strips of data, it is responsible for allocating the buffers. The application deallocates the buffers.

See the DG_IMAGE / DAT_JPEGCOMPRESSION operations.

Requirements to be a TWAIN-Compliant Source

Requirements

TWAIN-compliant Sources **must** support the following:

Operations

DG_CONTROL / DAT_CAPABILITY / MSG_GET
DG_CONTROL / DAT_CAPABILITY / MSG_GETCURRENT
DG_CONTROL / DAT_CAPABILITY / MSG_GETDEFAULT
DG_CONTROL / DAT_CAPABILITY / MSG_RESET
DG_CONTROL / DAT_CAPABILITY / MSG_SET

DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT

DG_CONTROL / DAT_IDENTITY / MSG_GET
DG_CONTROL / DAT_IDENTITY / MSG_OPENDS
DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS

DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER
DG_CONTROL / DAT_PENDINGXFERS / MSG_GET
DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET

DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET

DG_CONTROL / DAT_STATUS / MSG_GET

DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS
DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

DG_CONTROL / DAT_XFERGROUP / MSG_GET

DG_IMAGE / DAT_IMAGEINFO / MSG_GET

DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_GETDEFAULT
DG_IMAGE / DAT_IMAGELAYOUT / MSG_RESET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET

DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET

DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET

Capabilities

Every Source must support all five DG_CONTROL / DAT_CAPABILITY operations on:

CAP_XFERCOUNT

Every Source must support DG_CONTROL / DAT_CAPABILITY MSG_GET on:

CAP_SUPPORTEDCAPS
CAP_UICONTROLLABLE

Sources that supply image information must support DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULTon:

ICAP_COMPRESSION
ICAP_PLANARCHUNKY
ICAP_PHYSICALHEIGHT
ICAP_PHYSICALWIDTH
ICAP_PIXELFLAVOR

Sources that supply image information must support DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, MSG_RESET and MSG_SET on:

ICAP_BITDEPTH
ICAP_BITORDER
ICAP_PIXELTYPE
ICAP_UNITS
ICAP_XFERMECH
ICAP_XRESOLUTION
ICAP_YRESOLUTION

All Sources must implement the advertised features supported by their devices. They must make these features available to applications via the TWAIN protocol. For example, a Source that's connected to a device that has an ADF must support DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT on:

CAP_FEEDERENABLED
CAP_FEEDERLOADED

and DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, MSG_RESET and MSG_SET on:

CAP_AUTOFEED

If the ADF also supports ejecting and rewinding of pages then the Source should also support DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, MSG_RESET and MSG_SET on:

CAP_CLEARPAGE
CAP_REWINDPAGE

Other Topics

Custom Operations

Manufacturers may add custom operations to their Sources. These can also be made known to application manufacturers. This mechanism allows an application to access functionality not normally available from a generic TWAIN Source.

One use of this mechanism might be to implement device-specific diagnostics for a hardware diagnostic program. These custom operations should be used sparingly and never in place of pre-defined TWAIN operations.

Custom operations are defined by specifying special values for Data Groups (DGs), Data Argument Types (DATs), Messages (MSGs), and Capabilities (CAPs). The following areas have been reserved for custom definitions:

Data Groups	**Top 8 bit flags (bits 24 - 31) in the DG identifiers reserved for custom use.**
DATs	Designators with values greater than 8000 hex.
Messages	Designators with values greater than 8000 hex.
Capabilities	Designators with values greater than 8000 hex.

The responsibility for naming and managing the use of custom designators lies wholly upon the TWAIN element originating the designator and the element consuming it. Prior to interpreting a custom designator, the consuming element must check the originating element's ProductName string from its TW_IDENTITY structure. Since custom operation numbers may overlap, this is the only way to insure against confusion.

Custom Operations Enhancements

Checking the originating element's product name is not always practicable. Product names can change over time. It also forces a Source or Application to maintain a data base of TWAIN elements and the custom operations they support. Furthermore, a custom operation can only be used if the TWAIN developer has knowledge of it.

To solve all these problems, the TWAIN Working Group is proposing to create a Custom Operations Registry. Developers who want to use define a custom operation will need to register it with the TWAIN Working Group. Upon registration they will receive a unique set of values identifying the registered operation. Developers can register public and private operations. A complete description is needed to register a public operation. The complete list of custom operations will be periodically published by the Working Group. The list will contain the descriptions and values of all public custom operations. Private custom operations will appear with dummy names and no description.

Additionally the values ranging from 7000 to 7FFF hex are reserved for private custom operations used for testing and other purposes. Vendors are free to select any values inside this range for their private use.

Eventually, it is anticipated that custom operations that are of broad interest to TWAIN developers will be incorporated into the TWAIN spec.

Networking

If a Source supports connection to a remote device over a network, the Source is responsible for hiding the network dependencies of that device's operation from the application. The Source Manager does not know anything about networks.

In a networking situation, the Source will probably be built in two segments: One running on the machine local to the application, the other running remotely across the network. Sources are required to handle all the network interfacing with remote devices (real or logical) through local Source "stubs" that understand how to access both the network and the remote Source while interacting logically with the Source Manager.

The segment running on the local machine will probably be a "stub" Source. That is, the local stub will translate all operations received from the application and Source Manager into a form the remote source understands (that is, not necessarily TWAIN-defined operations). The stub also:

- Converts the information returned from the remote source into TWAIN-compliant results
- Handles local memory management for data copies and data transferring
- Isolates the network from the Source Manager and application
- Manages the connection with the remote Source
- Provides any needed code to handle local hardware (such as interface hardware)
- Provides the local user interface to control the remote Source

6

Entry Points and Triplet Components

Chapter Contents

Entry Points

TWAIN has two entry points:

- **DSM_Entry( )** - located in the Source Manager and typically called by applications. (The exceptions occur on the DG_CONTROL / DAT_NULL / MSG_CLOSEDSREQ and DG_CONTROL / DAT_NULL / MSG_XFERREADY operations. With these operations, a Windows Source calls the Source Manager to communicate to an application.)
- **DS_Entry( )** - located in the Source and called only by the Source Manager.

Programming Basics

- Upon entry, the parameters must be ordered on the stack in Pascal form. Be sure that your code expects this ordering rather than the reverse order that C uses.
- The keyword FAR is included in the entry point syntax to accommodate the Windows/DOS segmented addressing scheme. It has no value for Macintosh or UNIX systems so the TWAIN.H file simply defines FAR as an empty value for Macintosh and UNIX.

Declaration of DSM_Entry( )

Written in C code form, the declaration looks like this:

On Windows

```
TW_UINT16 FAR PASCAL DSM_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       pTW_IDENTITY   pDest,      // destination of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

On Macintosh

```
FAR PASCAL TW_UINT16 DSM_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       pTW_IDENTITY   pDest,      // destination of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

Parameters of DSM_Entry( )

pOrigin

This points to a TW_IDENTITY structure, allocated by the application, that describes the application making the call. One of the fields in this structure, called Id, is an arbitrary and unique identifier assigned by the Source Manager to tag the application as a unique TWAIN entity. The Source Manager maintains a copy of the application's identity structure, so the application must not modify that structure unless it first breaks its connection with the Source Manager, then reconnects to cause the Source Manager to store the new, modified identity.

pDest

This is set either to NULL if the application is aiming the operation at the Source Manager or to the TW_IDENTITY structure of the Source that the application is attempting to reach. The application allocated the space for the Source's identity structure when it decided which Source was to be connected. The Source's TW_IDENTITY.Id is also uniquely set by the Source Manager when the Source is opened and should not be modified by the Source. The application should not count on the value of this field being consistent from one session to the next because the Source Manager reallocates these numbers every time it is opened. The Source Manager keeps a copy of the Source's identity structure as should the application and the Source.

DG

The Data Group of the operation triplet. Currently, only DG_CONTROL and DG_IMAGE are defined.

DAT

The Data Argument Type of the operation triplet. A complete list appears later in this chapter.

MSG

The Message of the operation triplet. A complete list appears later in this chapter.

pData

The pData parameter is of type TW_MEMREF and is a pointer to the data (a variable or, more typically, a structure) that will be used according to the action specified by the operation triplet.

Declaration of DS_Entry( )

DS_Entry is only called by the Source Manager. Written in C code form, the declaration looks like this:

On Windows

```
TW_UINT16 FAR PASCAL DSM_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

On Macintosh

```
FAR PASCAL TW_UINT16 DSM_Entry
     ( pTW_IDENTITY   pOrigin,    // source of message
       TW_UINT32      DG,         // data group ID: DG_xxxx
       TW_UINT16      DAT,        // data argument type: DAT_xxxx
       TW_UINT16      MSG,        // message ID: MSG_xxxx
       TW_MEMREF      pData       // pointer to data
     );
```

Data Groups

TWAIN operations can be broadly classified into two data groups:

Control Oriented (DG_CONTROL)

Controls the TWAIN session

Consumed by both Source Manager and Source

Data Oriented (DG_IMAGE)

Indicates the kind of data to be transferred.

Currently, only image data is supported but this could be expanded to include text, etc. This has several future implications. If more than one data type exists, an application and a Source will need to decide what type(s) of data the Source can, and will be allowed to, produce before a transfer can occur. Further, if multiple transfers are being generated from a single acquisition—such as when image and text are intermixed and captured from the same page—it must be unambiguous which type of data is being returned from each data transfer.

Programming Basics

Note the following:

- Data Group designators are 32-bit, unsigned values. The actual values that are assigned are powers of two (bit flags) so that the DGs can be easily masked.
- There are 24 DGs designated as "reserved" for pre-defined DGs . Two are currently in use. The top 8 bits are reserved for custom DGs.

Data Argument Types

Data Argument Types, or DATs, are used to allow programmatic identification of the TWAIN type for the structure of status variable referenced by the entry point parameter pData. pData will always point to a variable or data structure defined by TWAIN. If the consuming application or Source switches (cases, etc.) on the DAT specified in the formal parameter list of the entry point call, it can handle the form of the referenced data correctly.

Table 6-1. Data Argument Types

Data Type	Used by	Associated structure or type
DAT_NULL	ANY DG	Null structure. No data required for the operation
DAT_CAPABILITY	DG_CONTROL	TW_CAPABILITY structure
DAT_EVENT	DG_CONTROL	TW_EVENT structure
DAT_IDENTITY	DG_CONTROL	TW_IDENTITY structure
DAT_PARENT	DG_CONTROL	TW_INT32 On Windows - low word=Window handle On Macintosh - Set to NULL
DAT_PENDINGXFERS	DG_CONTROL	TW_PENDINGXFERS structure
DAT_SETUPFILEXFER	DG_CONTROL	TW_SETUPFILEXFER structure
DAT_SETUPMEMXFER	DG_CONTROL	TW_SETUPMEMXFER structure
DAT_STATUS	DG_CONTROL	TW_STATUS structure
DAT_USERINTERFACE	DG_CONTROL	TW_USERINTERFACE structure
DAT_XFERGROUP	DG_CONTROL	TW_UINT32 A DG designator describing data to be transferred (currently only image data is supported)
DAT_CIECOLOR	DG_IMAGE	TW_CIECOLOR structure
DAT_GRAYRESPONSE	DG_IMAGE	TW_GRAYRESPONSE structure
DAT_IMAGEFILEXFER	DG_IMAGE	Operates on NULL data. Filename/Format already negotiated
DAT_IMAGEINFO	DG_IMAGE	TW_IMAGEINFO structure
DAT_IMAGELAYOUT	DG_IMAGE	TW_IMAGELAYOUT structure
DAT_IMAGEMEMXFER	DG_IMAGE	TW_IMAGEMEMXFER structure
DAT_IMAGENATIVEXFER	DG_IMAGE	TW_UINT32; On Windows - low word=DIB handle On Macintosh - PicHandle
DAT_JPEGCOMPRESSION	DG_IMAGE	TW_JPEGCOMPRESSION structure
DAT_PALETTE8	DG_IMAGE	TW_PALETTE8 structure
DAT_RGBRESPONSE	DG_IMAGE	TW_RGBRESPONSE structure

Messages

A Message, or MSG, is used to communicate between TWAIN elements what action is to be taken upon a particular piece of data, or for a data-less operation, what action to perform. If an application wants to make anything happen in, or inquire any information from, a Source or the Source Manager, it must make a call to DSM_Entry( ) with the proper MSG as one parameter of the operation triplet. The data to be acted upon is also specified in the parameter list of this call.

A MSG is always associated with a Data Group (DG) identifier and a Data Argument Type (DAT) identifier in an operation triplet. This operation unambiguously specifies what action is to be taken on what data. Refer to Chapter 7 for the list of defined operation triplets.

Table 6-2. Messages

Message ID	Valid DAT(s)	Description of Specified Action
MSG_NULL	None	No action to be taken
MSG_GET	various DATs	Get all Available Values including Current & Default
MSG_GETCURRENT	various DATs	Get Current value
MSG_GETDEFAULT	various DATs	Get Source's preferred default value
MSG_RESET	various DATs	Return specified item to power-on (TWAIN default) condition
MSG_SET	various DATs	Set one or more values
MSG_CLOSEDS	DAT_IDENTITY	Close the specified Source
MSG_CLOSEDSM	DAT_PARENT	Close the Source Manager
MSG_CLOSEDSREQ	DAT_NULL	Source requests for application to close Source
MSG_DISABLEDS	DAT_USERINTERFACE	Disable data transfer in the Source
MSG_ENABLEDS	DAT_USERINTERFACE	Enable data transfer in the Source
MSG_ENDXFER	DAT_PENDINGXFERS	Application tells Source that transfer is over
MSG_GETFIRST	DAT_IDENTITY	Get first element from a "list"
MSG_GETNEXT	DAT_IDENTITY	Get next element from a "list"
MSG_OPENDS	DAT_IDENTITY	Open and Initialize the specified Source
MSG_OPENDSM	DAT_PARENT	Open the Source Manager
MSG_PROCESSEVENT	DAT_EVENT	Tells Source to check if event/message belongs to it
MSG_USERSELECT	DAT_IDENTITY	Presents dialog of all Sources to select from
MSG_XFERREADY	DAT_NULL	The Source has data ready for transfer to the application

Custom Components of Triplets

Custom Data Groups

A manufacturer may choose to implement custom data descriptors that require a new Data Group. This would be needed if someone decides to extend TWAIN to, say, satellite telemetry.

- The top 8 bits of every DG_xxxx identifier are reserved for use as custom DGs. Custom DG identifiers must use one of the upper 8 bits of the DG_xxxx identifier. Remember, DGs are bit flags.
- The originator of the custom DG must fill the ProductName field in the application or Source's TW_IDENTITY structure with a uniquely descriptive name. The consumer will look at this field to determine whose custom DG is being used.
- TWAIN provides no formal allocation (or vendor-specific "identifier blocks") for custom data group identifiers nor does it do any coordination to avoid collisions.
- The DG_CUSTOMBASE value resides in the TWAIN.H file. All custom IDs must be numerically greater than this base. A similar custom base "address" is defined for Data Argument Types, Messages, Capabilities, Return Codes, and Condition Codes. The only difference in concept is that DGs are the only designators defined as bit flags. All other custom values can be any integer value larger than the xxx_CUSTOMBASE defined for that type of designator.

Custom Data Argument Types

DAT_CUSTOMBASE is defined in the TWAIN.H file to allow a Source vendor to define "custom" DATs for their particular device(s). The application can recognize the Source by checking the TW_IDENTITY.ProductName and the TW_IDENTITY.TW_VERSION structure. If an application is aware that this particular Source offers custom DATs, it can use them. No changes to TWAIN or the Source Manager are required to support such identifiers (or the data structures which they imply).

Refer to the TWAIN.H file for the value of DAT_CUSTOMBASE for custom DATs. All custom values must be numerically greater than this constant.

Custom Messages

As with the DATs, MSG_CUSTOMBASE is included in TWAIN.H so that the Source writer can create custom messages specific to their Source. If the applications understand these custom messages, actions beyond those defined in this specification can be performed through the normal TWAIN mechanism. No modifications to TWAIN or the Source Manager are required.

Remember that the consumer of these custom values will look in your TW_IDENTITY.ProductName field to clarify what the identifier's value means—there is no other protection for overlapping custom definitions. Refer to the TWAIN.H file for the value of MSG_CUSTOMBASE for custom Messages. All custom values must be numerically greater than this value.

7

Operation Triplets

Chapter Contents

An Overview of the Triplets

From Application to Source Manager (Control Information)

Data Group	Data Argument Type	Message	Page #
DG_CONTROL	DAT_IDENTITY	MSG_CLOSEDS	7-17
		MSG_GETDEFAULT	7-20
		MSG_GETFIRST	7-21
		MSG_GETNEXT	7-22
		MSG_OPENDS	7-23
		MSG_USERSELECT	7-25
DG_CONTROL	DAT_PARENT	MSG_CLOSEDSM	7-29
		MSG_OPENDSM	7-30
DG_CONTROL	DAT_STATUS	MSG_GET	7-40

From Application to Source (Control Information)

Data Group	Data Argument Type	Message	Page #
DG_CONTROL	DAT_CAPABILITY	MSG_GET	7-133
		MSG_GETCURRENT	7-135
		MSG_GETDEFAULT	7-137
		MSG_QUERYSUPPORT	7-139
		MSG_RESET	7-141
		MSG_SET	7-143
DG_CONTROL	DAT_EVENT	MSG_PROCESSEVENT	7-146
DG_CONTROL	DAT_PENDINGXFERS	MSG_ENDXFER	7-
		MSG_GET	7-
		MSG_RESET	7-
DG_CONTROL	DAT_SETUPFILEXFER	MSG_GET	7-
		MSG_GETDEFAULT	7-
		MSG_RESET	7-
		MSG_SET	7-
DG_CONTROL	DAT_SETUPMEMXFER	MSG_GET	7-
DG_CONTROL	DAT_STATUS	MSG_GET	7-
DG_CONTROL	DAT_USERINTERFACE	MSG_DISABLEDS	7-
		MSG_ENABLEDS	7-
DG_CONTROL	DAT_XFERGROUP	MSG_GET	7-

From Application to Source (Image Information)

Data Group	Data Argument Type	Message	Page #
DG_IMAGE	DAT_CIECOLOR	MSG_GET	7-47
DG_IMAGE	DAT_GRAYRESPONSE	MSG_RESET	7-48
		MSG_SET	7-49
DG_IMAGE	DAT_IMAGEFILEXFER	MSG_GET	7-50
DG_IMAGE	DAT_IMAGEINFO	MSG_GET	7-52
DG_IMAGE	DAT_IMAGELAYOUT	MSG_GET	7-53
		MSG_GETDEFAULT	7-55
		MSG_RESET	7-56
		MSG_SET	7-57
DG_IMAGE	DAT_IMAGEMEMXFER	MSG_GET	7-59
DG_IMAGE	DAT_IMAGENATIVEXFER	MSG_GET	7-61
DG_IMAGE	DAT_JPEGCOMPRESSION	MSG_GET	7-63
		MSG_GETDEFAULT	7-64
		MSG_RESET	7-65
		MSG_SET	7-66
DG_IMAGE	DAT_PALETTE8	MSG_GET	7-67
		MSG_GETDEFAULT	7-68
		MSG_RESET	7-69
		MSG_SET	7-70

DG_IMAGE	DAT_RGBRESPONSE	MSG_RESET	7-71
		MSG_SET	7-72

From Source Manager to Source (Control Information)

Data Group	Data Argument Type	Message	Page #
DG_CONTROL	DAT_IDENTITY	MSG_CLOSEDS	7-148
		MSG_GET	7-150
		MSG_OPENDS	7-

From Source to Application (Control Information via the Source Manager)

(Used by Windows Sources only)

Data Group	Data Argument Type	Message	Page #
DG_CONTROL	DAT_NULL	MSG_CLOSEDSREQ	7-27
		MSG_XFERREADY	7-28

Format of the Operation Triplet Descriptions

The following pages describe the operation triplets. They are all included and are arranged in alphabetical order using the Data Group, Data Argument Type, and Message identifier list.

There are three operations that are duplicated because that have a different originator and/or destination in each case. They are:

- DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS
  - ✓ from Application to Source Manager
  - ✓ from Source Manager to Source
- DG_CONTROL / DAT_IDENTITY / MSG_OPENDS
  - ✓ from Application to Source Manager
  - ✓ from Source Manager to Source
- DG_CONTROL / DAT_STATUS / MSG_GET
  - ✓ from Application to Source Manager
  - ✓ from Application to Source

The format of each page is:

Triplet - The concise DG / DAT / MSG information

Call

Actual format of the routine call (parameter list) for the operation. Identification of the data structure used for the pData parameter is included.

Valid States

The states in which the application, Source Manager, or Source may legally invoke the operation.

Description

General description of the operation.

Origin of the Operation (Application, Source Manager or, Source)

The action(s) the application, Source Manager, or Source should take before invoking the operation.

Destination of the Operation (Source Manager or Source)

The action that the destination element (Source Manager or Source) of the operation will take.

Return Codes

The Return Codes and Condition Codes that are defined and valid for this operation.

See Also

Lists other related operation triplets, capabilities, constants, etc.

Operation Triplets

DG_CONTROL / DAT_CAPABILITY / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_CAPABILITY, MSG_GET, pCapability);
```

pCapability = A pointer to a TW_CAPABILITY structure

Valid States

4 through 7

Description

Returns the Source's Current, Default and Available Values for a specified capability.

These values reflect previous MSG_SET operations on the capability, or Source's automatic changes. (See MSG_SET).

Note: This operation does not change the Current or Available Values of the capability.

Application

Set the pCapability fields as follows:

```
pCapability->Cap = the CAP_xxx or ICAP_xxx identifier
pCapability->ConType = TWON_DONTCARE16
pCapability->hContainer = NULL
```

The Source will allocate the memory for the necessary container structure but the application must free it when the operation is complete and the application no longer needs to maintain the information.

Use MSG_GET:

- As the first step in negotiation of a capability's Available Values.
- To check the results if a MSG_SET returns TWRC_CHECKSTATUS.
- To check the Available, Current and Default Values with one command.

This operation may fail for a low memory condition. Either recover from a TWCC_LOWMEMORY failure by freeing memory for the Source to use so it can continue, or terminating the acquisition and notifying the user of the low memory problem.

Source

If the application requests this operation on a capability your Source does not recognize (and you're sure you've implemented all the capabilities that you're required to), disregard the operation, but return TWRC_FAILURE with TWCC_BADCAP.

If you support the capability, fill in the fields listed below and allocate the container structure and place its handle into pCapability->hContainer. The container should be referenced by a "handle" of type TW_HANDLE.

Fill the fields in pCapability as follows:

```
pCapability->ConType = TWON_ARRAY,
TWON_ONEVALUE,
TWON_ENUMERATION, or
TWON_RANGE

pCapability->hContainer = TW_HANDLE referencing a container of ConType
```

Set ConType to the container type your Source uses for this capability. For container types of TWON_ARRAY and TWON_ONEVALUE provide the Current Value. For container types TWON_ENUMERATION and TWON_RANGE provide the Current, Default and Available Values.

This is a memory allocation operation. It is possible for this operation to fail due to a low memory condition. Be sure to verify that the allocation is successful. If it is not, attempt to reduce the amount of memory occupied by the Source. If the allocation cannot be made, return TWRC_FAILURE with TWCC_LOWMEMORY to the application and set the pCapability->hContainer handle to NULL.

Note that the Source **must** be able to respond to an inquiry about any of its capabilities at **any time** that the Source is open.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
  TWCC_BADCAP        /* Unknown capability--Source does not */
                     /* does not recognize this capability  */
  TWCC_BADDEST       /* No such Source in session with      */
                     /* application                         */
  TWCC_LOWMEMORY     /* Not enough memory to complete the   */
                     /* operation                           */
  TWCC_SEQERROR      /* Operation invoked in invalid state  */
```

See Also

- DG_CONTROL / DAT_CAPABILITY / MSG_GETCURRENT, MSG_GETDEFAULT, MSG_RESET, and MSG_SET
- Capability Constants (in Chapter 8)
- Capability Containers: TW_ONEVALUE, TW_ENUMERATION, TW_RANGE, TW_ARRAY (in Chapter 8)
- Listing of all capabilities (in Chapter 9)

DG_CONTROL / DAT_CAPABILITY / MSG_GETCURRENT

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_CAPABILITY, MSG_GETCURRENT,
pCapability);
```

pCapability = A pointer to a TW_CAPABILITY structure

Valid States

4 through 7

Description

Returns the Source's Current Value for the specified capability.

The Current Value reflects previous MSG_SET operations on the capability, or Source's automatic changes. (See MSG_SET).

Note: This operation does not change the Current Values of the capability.

Application

Set the pCapability fields as follows:

```
pCapability->Cap = the CAP_xxx or ICAP_xxx identifier
pCapability->ConType = TWON_DONTCARE16
pCapability->hContainer = NULL
```

The Source will allocate the memory for the necessary container structure but the application must free it when the operation is complete and the application no longer needs to maintain the information.

Use MSG_GETCURRENT:

- To check the Source's power-on Current Values (see Chapter 9 for TWAIN-defined defaults for each capability).
- To check just the Current Value (in place of using MSG_GET).
- In State 6 to determine the settings. They could have been set by the user (if TW_USERINTERFACE.ShowUI = TRUE) or be the results of automatic processes used by the Source.

This operation may fail for a low memory condition. Either recover from a TWCC_LOWMEMORY failure by freeing memory for the Source to use so it can continue, or terminating the acquisition and notifying the user of the low memory problem.

Source

If the application requests this operation on a capability your Source does not recognize (and you're sure you've implemented all the capabilities that you're required to), disregard the operation, but return TWRC_FAILURE with TWCC_BADCAP.

If you support the capability, fill in the fields listed below and allocate the container structure and place its handle into pCapability->hContainer. The container should be referenced by a "handle" of type TW_HANDLE.

Fill the fields in pCapability as follows:

```
pCapability->ConType = TWON_ARRAY or TWON_ONEVALUE
pCapability->hContainer = TW_HANDLE referencing a container of ConType
```

Set ConType to the container type that matches the type for this capability. Fill the fields in the container structure with the Current Value of the capability.

This is a memory allocation operation. It is possible for this operation to fail due to a low memory condition. Be sure to verify that the allocation is successful. If it is not, attempt to reduce the amount of memory occupied by the Source. If the allocation cannot be made, return TWRC_FAILURE with TWCC_LOWMEMORY to the application and set the pCapability->hContainer handle to NULL.

Note that the Source **must** be able to respond to an inquiry about any of its capabilities at **any time** that the Source is open.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
  TWCC_BADCAP         /* unknown capability--Source does not  */
                      /* recognize this capability            */
  TWCC_BADDEST        /* No such Source in-session with       */
                      /* application                          */
  TWCC_LOWMEMORY      /* Not enough memory to complete the    */
                      /* operation                            */
  TWCC_SEQERROR       /* Operation invoked in invalid  state  */
```

See Also

DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETDEFAULT, MSG_RESET, and MSG_SET

Capability Constants (in Chapter 8)

Capability Containers: TW_ONEVALUE, TW_ENUMERATION, TW_RANGE, TW_ARRAY (in Chapter 8)

Listing of all capabilities (in Chapter 9)

DG_CONTROL / DAT_CAPABILITY / MSG_GETDEFAULT

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_CAPABILITY, MSG_GETDEFAULT,
pCapability);
```

pCapability = A pointer to a TW_CAPABILITY structure

Valid States

4 through 7

Description

Returns the Source's Default Value. This is the Source's preferred default value.

The Source's Default Value cannot be changed.

Application

Set the pCapability fields as follows:

```
pCapability->Cap = the CAP_xxx or ICAP_xxx identifier
pCapability->ConType = TWON_DONTCARE16
pCapability->hContainer = NULL
```

The Source will allocate the memory for the necessary container structure but the application must free it when the operation is complete and the application no longer needs to maintain the information.

Use MSG_GETDEFAULT:

- To check the Source's preferred Values. Using the Source's preferred default as the Current Value may increase performance in some Sources.

This operation may fail for a low memory condition. Either recover from a TWCC_LOWMEMORY failure by freeing memory for the Source to use so it can continue, or terminating the acquisition and notifying the user of the low memory problem.

Source

If the application requests this operation on a capability your Source does not recognize (and you are sure you have implemented all the capabilities that you're required to), disregard the operation, but return TWRC_FAILURE with TWCC_BADCAP.

If you support the capability, fill in the fields listed below and allocate the container structure and place its handle into pCapability->hContainer. The container should be referenced by a "handle" of type TW_HANDLE.

Fill the fields in pCapability as follows:

```
pCapability->ConType = TWON_ARRAY or TWON_ONEVALUE
pCapability->hContainer = TW_HANDLE referencing a container of ConType
```

Set ConType to the container type that matches for this capability. Fill the fields in the container with the Default Value of this capability.

The Default Value is the preferred value for the Source. This value is used as the power-on value for capabilities if TWAIN does not specify a default.

This is a memory allocation operation. It is possible for this operation to fail due to a low memory condition. Be sure to verify that the allocation is successful. If it is not, attempt to reduce the amount of memory occupied by the Source. If the allocation cannot be made return TWRC_FAILURE with TWCC_LOWMEMORY to the application and set the pCapability->hContainer handle to NULL.

Note that the Source **must** be able to respond to an inquiry about any of its capabilities at **any time** that the Source is open.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
  TWCC_BADCAP         /* unknown capability--Source does not */
                      /* recognize this capability           */
  TWCC_BADDEST        /* No such Source in-session with      */
                      /* application                         */
  TWCC_LOWMEMORY      /* Not enough memory to complete the   */
                      /* operation                           */
  TWCC_SEQERROR       /* Operation invoked in invalid state  */
```

See Also

DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_RESET, and MSG_SET

Capability Constants (in Chapter 8)

Capability Containers: TW_ONEVALUE, TW_ENUMERATION, TW_RANGE, TW_ARRAY (in Chapter 8)

Listing of all capabilities (in Chapter 9)

DG_CONTROL / DAT_CAPABILITY / MSG_QUERYSUPPORT

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_CAPABILITY, MSG_GETDEFAULT,
pCapability);
```

pCapability = A pointer to a TW_CAPABILITY structure

Valid States

4 through 7

Description

Returns the Source's support status of this capability.

Application

Set the pCapability fields as follows:

```
pCapability->Cap = the CAP_xxx or ICAP_xxx identifier
pCapability->ConType = TWON_ONEVALUE
pCapability->hContainer = NULL
```

The Source will allocate the memory for the necessary container structure but the application must free it when the operation is complete and the application no longer needs to maintain the information.

Use MSG_QUERYSUPPORT:

- To check the whether the Source supports a particular operation on the capability.

This operation may fail for a low memory condition. Either recover from a TWCC_LOWMEMORY failure by freeing memory for the Source to use so it can continue, or terminating the acquisition and notifying the user of the low memory problem.

Source

If the application requests this operation on a capability your Source does not recognize (and you're sure you've implemented all the capabilities that you're required to), do not disregard the operation, but fill out the TWON_ONEVALUE container with

Fill the fields in pCapability as follows:

```
pCapability->ConType = TWON_ONEVALUE
```

```
pCapability->hContainer = TW_HANDLE referencing a container of type
TW_ONEVALUE.
```

Fill the fields in TW_ONVALUE as follows:

ItemType = TWTW_INT32;

Item = Bit pattern representing the set of operation that are supported by the Data Source on this capability (TWQC_GET, TWQC_SET, TWQC_GETDEFAULT, TWQC_RESET);

This is a memory allocation operation. It is possible for this operation to fail due to a low memory condition. Be sure to verify that the allocation is successful. If it is not, attempt to reduce the amount of memory occupied by the Source. If the allocation cannot be made return TWRC_FAILURE with TWCC_LOWMEMORY to the application and set the pCapability->hContainer handle to NULL.

Note that the Source **must** be able to respond to an inquiry about any of its capabilities at **any time** that the Source is open.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
TWCC_BADDEST          /* No such Source in-session with      */
                      /* application                         */
TWCC_LOWMEMORY        /* Not enough memory to complete the   */
                      /* operation                           */
```

See Also

- DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_RESET, and MSG_SET
- Capability Constants (in Chapter 8)
- Capability Container: TW_ONEVALUE (in Chapter 8).
- Listing of all capabilities (in Chapter 9)

DG_CONTROL / DAT_CAPABILITY / MSG_RESET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_CAPABILITY, MSG_RESET, pCapability);
```

pCapability = A pointer to a TW_CAPABILITY structure

Valid States

4 only

Description

Change the Current Value of the specified capability back to its power-on value and return the new Current Value.

The power-on value is the Current Value the Source started with when it entered State 4 after a DG_CONTROL / DAT_IDENTITY / MSG_OPENDS. These values are listed as TWAIN defaults (in Chapter 9). If "no default" is specified, the Source uses it preferred default value (returned from MSG_GETDEFAULT).

Application

Set the pCapability fields as follows:

```
pCapability->Cap = the CAP_xxx or ICAP_xxx identifier
pCapability->ConType = TWON_DONTCARE16
pCapability->hContainer = NULL
```

The Source will allocate the memory for the necessary container structure but the application must free it when the operation is complete and the application no longer needs to maintain the information.

Use MSG_RESET:

- To set the Current Value of the specified capability back to its power-on value.

This operation may fail for a low memory condition. Either recover from a TWCC_LOWMEMORY failure by freeing memory for the Source to use so it can continue, or terminating the acquisition and notifying the user of the low memory problem.

Source

If the application requests this operation on a capability your Source does not recognize (and you're sure you've implemented all the capabilities that you're required to), disregard the operation, but return TWRC_FAILURE with TWCC_BADCAP.

If you support the capability, reset the Current Value of the capability back to its power-on value. This value must also match the TWAIN default listed in Chapter 9.

Also return the new Current Value (just like in a MSG_GETCURRENT). Fill in the fields listed below and allocate the container structure and place its handle into pCapability->hContainer. The container should be referenced by a "handle" of type TW_HANDLE.

Fill the fields in pCapability as follows:

```
pCapability->ConType = TWON_ARRAY or TWON_ONEVALUE
pCapability->hContainer = TW_HANDLE referencing a container of ConType
```

Set ConType to the container type that matches the type for this capability. Fill the fields in the container structure with the Current Value of the capability (after resetting it as stated above).

This is a memory allocation operation. It is possible for this operation to fail due to a low memory condition. Be sure to verify that the allocation is successful. If it is not, attempt to reduce the amount of memory occupied by the Source. If the allocation cannot be made return TWRC_FAILURE with TWCC_LOWMEMORY to the application and set the pCapability->hContainer handle to NULL.

Note that this operation is **only** valid in State 4, unless CAP_EXTENDEDCAPS was negotiated. Any attempt to invoke it in any other state should be disregarded, though the Source should return TWRC_FAILURE with TWCC_SEQERROR.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
  TWCC_BADCAP       /* unknown capability--Source does not  */
                    /* recognize this capability            */
  TWCC_BADDEST      /* No such Source in-session with       */
                    /* application                          */
  TWCC_LOWMEMORY    /* Not enough memory to complete the    */
                    /* operation                            */
  TWCC_SEQERROR     /* Operation invoked in invalid state   */
```

See Also

- DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, and MSG_SET
- Capability Constants (in Chapter 8)
- Capability Containers: TW_ONEVALUE, TW_ENUMERATION, TW_RANGE, TW_ARRAY (in Chapter 8)
- Listing of all capabilities (in Chapter 9)

DG_CONTROL / DAT_CAPABILITY / MSG_SET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_CAPABILITY, MSG_SET, pCapability);
```

pCapability = A pointer to a TW_CAPABILITY structure

Valid States

4 only (During State 4, applications can also negotiate with Sources for permission to set the value(s) of specific capabilities in States 5 and 6 through CAP_EXTENDEDCAPS.)

Description

Changes the Current Value(s) and Available Values of the specified capability to those specified by the application.

Current Values are set when the container is a TW_ONEVALUE or TW_ARRAY. Available and Current Values are set when the container is a TW_ENUMERATION or TW_RANGE.

Note: Sources are not required to allow restriction of their Available Values, however, this is strongly recommended.

Application

An application will use the setting of a capability's Current and Available Values differently depending on how the Source was enabled (DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS).

If TW_USERINTERFACE.ShowUI = TRUE

- In State 4, set the Current Value to be displayed to the user as the current value. This value will be used for acquiring the image unless changed by the user or an automatic process (such as ICAP_AUTOBRIGHT).
- In State 4, set the Available Values to restrict the settings displayed to the user and available for use by the Source.
- In State 6, get the Current Value which was chosen by the user or automatic process. This is the setting used in the upcoming transfer.

If TW_USERINTERFACE.ShowUI = FALSE

- In State 4, set the Current Value to the setting that will be used to acquire images (unless automatic settings are set to TRUE, for example: ICAP_AUTOBRIGHT).
- In State 6, get the Current Value which was chosen by any automatic processes. This is the setting used in the upcoming transfer.

If possible, use the same container type in a MSG_SET that the Source returned from a MSG_GET. Allocate the container structure. This is where you will place the value(s) you wish to have the Source set. Store the handle into pCapability->hContainer. The container must be referenced by a "handle" of type TW_HANDLE.

Set the following:

```
pCapability->ConType = TWON_ARRAY,
                       TWON_ONEVALUE,
                       TWON_ENUMERATION, or
                       TWON_RANGE

pCapability->Cap     =  CAP_xxxx designator of
                        capability of interest

pCapability->hContainer = TW_HANDLE referencing a
                          container of ConType
```

Place the value(s) that you wish the Source to use in the container. If successful, these values will supersede any previous negotiations for this capability.

The application must free the container it allocated when the operation is complete and the application no longer needs to maintain the information.

Source

Return TWRC_FAILURE / TWCC_BADCAP:

- If the application requests this operation on a capability your Source does not recognize (and you're sure you've implemented all the capabilities that you're required to). Disregard the operation.

Return TWRC_FAILURE / TWCC_BADVALUE:

- If the application requests that a value be set that lies outside the supported range of values for the capability (smaller than your minimum value or larger than your maximum value). Set the value to that which most closely approximates the requested value.
- If the application sends a container that you do not support, or do not support in a MSG_SET.
- If the application attempts to set the Available Values and the Source does not support restriction of this capability's Available Values.

Return TWRC_CHECKSTATUS:

- If the application requests one or more values that lie within the supported range of values (but that value does not exactly match one of the supported values), set the value to the nearest supported value. The application should then do a MSG_GET to check these values.

Return TWRC_FAILURE / TWCC_SEQERROR:

- If the application sends the MSG_SET outside of State 4 and the capability has not been negotiated in CAP_EXTENDEDCAPS.

If the request is acceptable, use the container structure referenced by pCapability->hContainer to set the Current and Available Values for the capability. If the container type is TWON_ONEVALUE or TWON_ARRAY, set the Current Value for the capability to that value. If the container type is TWON_RANGE or TWON_ENUMERATION, the Source will **optionally** limit the Available Values for the capability to match those provided by the application, masking all other internal values so that the user cannot select them. Though this behavior is not mandatory, it is strongly encouraged.

Important Note: Sources should accommodate requests to limit Available Values. In the interest of adoptability for the breadth of Source manufacturers, such accommodation is not required. It is recommended, however, that the Sources do so, and that the Source's user interface be modified (from its power-on state, and when the user interface is raised) to reflect any limitation of choices implied by the newly negotiated settings.

For example, if an application can only accept black and white image data, it tells the Source of this limitation by doing a MSG_SET on ICAP_PIXELTYPE with a TW_ENUMERATION or TW_RANGE container containing only TWPT_BW (black and white).

If the Source disregards this negotiated value and fails to modify its user interface, the user may select to acquire a color image. Either the user's selection would fail (for reasons unclear to the user) or the transfer would fail (also for unclear reasons for the user). The Source should strive to prevent such situations.

Return Codes

```
TWRC_SUCCESS
TWRC_CHECKSTATUS      /* capability value(s) could not be */
                      /* matched exactly                  */

TWRC_FAILURE
   TWCC_BADCAP        /* unknown capability--Source does  */
                      /* not recognize this capability    */
   TWCC_BADDEST       /* No such Source in-session with   */
                      /* application                      */
   TWCC_BADVALUE      /* illegal value(s)--outside        */
                      /* Source's range for capability    */
   TWCC_SEQERROR      /* Operation invoked in invalid     */
                      /* state                            */
```

See Also

- DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, MSG_GETDEFAULT, and MSG_RESET
- Capability Constants (in Chapter 8)
- Capability Containers: TW_ONEVALUE, TW_ENUMERATION, TW_RANGE, TW_ARRAY (in Chapter 8)
- Listing of all capabilities (in Chapter 9)

DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_EVENT, MSG_PROCESSEVENT, pEvent);
```

pEvent = A pointer to a TW_EVENT structure

Valid States

5 through 7

Description:

This operation supports the distribution of events from the application to Sources so that the Source can maintain its user interface and return messages to the application. Once the application has enabled the Source, it **must immediately** begin sending to the Source all events that enter the application's main event loop. This allows the Source to update its user interface in real-time and to return messages to the application which cause state transitions. Even if the application overrides the Source's user interface, it must forward all events once the Source has been enabled. The Source will tell the application whether or not each event belongs to the Source.

Note: Events only need to be forwarded to the Source while it is enabled.

The Source should be structured such that identification of the event's "owner" is handled before doing anything else. Further, the Source should return **immediately** if the Source isn't the owner. This convention should minimize performance concerns for the application (remember, these events are only sent while a Source is enabled—that is, just before and just after the transfer is taking place).

Application

Make pEvent->pEvent point to the EventRecord (on Macintosh) or message structure (on Windows).

Note: On return, the application should check the Return Code from DSM_Entry() for TWRC_DSEVENT or TWRC_NOTDSEVENT. If TWRC_DSEVENT is returned, the application should not process the event—it was consumed by the Source. If TWRC_NOTDSEVENT is returned, the application should process the event as it normally would.

With either of these Return Codes, the application should also check the pEvent->TWMessage and switch on the result. This is the mechanism used by the Source to notify the application that a data transfer is ready or that it should close the Source. The Source can return one of the following messages:

```
MSG_XFERREADY   /* Source has one or more images */
                /* ready to transfer             */
MSG_CLOSEDSREQ  /* Source wants to be closed,    */
                /* usually initiated by a        */
                /* user-generated event          */
MSG_NULL        /* no message for application    */
```

Source

Process this operation **immediately** and return to the application **immediately** if the event doesn't belong to you. Be aware that the application will be sending thousands of messages to you. Consider in-line processing and global flags to speed implementation.

Return Codes

```
TWRC_DSEVENT          /* Source consumed event--application*/
                      /* should not process it     */
TWRC_NOTDSEVENT       /* Event belongs to application -  */
                      /* process as usual             */
TWRC_FAILURE
    TWCC_BADDEST      /* No such Source in-session      */
                      /* with application               */
    TWCC_SEQERROR     /* Operation invoked in invalid   */
                      /* state                          */
```

See Also

- DG_CONTROL / DAT_NULL / MSG_CLOSEDSREQ
  DG_CONTROL / DAT_NULL / MSG_XFERREADY
- Event loop information (in Chapter 3)

DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS *(from Application to Source Manager)*

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_IDENTITY, MSG_CLOSEDS,
pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure

Valid States

4 only (Transitions to State 3, if successful)

Description

When an application is finished with a Source, it must formally close the session between them using this operation. This is necessary in case the Source only supports connection with a single application (many desktop scanners will behave this way). A Source such as this cannot be accessed by other applications until its current session is terminated.

Application

Reference pSourceIdentity to the application's copy of the TW_IDENTITY structure for the Source whose session is to be ended. The application needs to unload the Source from memory after it is closed. The process for unloading the Source is similar to that used to unload the Source Manager.

Source Manager

On Macintosh only—Closes the Source and removes it from memory, following receipt of TWRC_SUCCESS from the Source.

On Windows only—Checks its internal counter to see whether any other applications are accessing the specified Source. If so, the Source Manager takes no other action. If the closing application is the last to be accessing this Source, the Source Manager closes the Source (forwards this triplet to it) and removes it from memory, following receipt of TWRC_SUCCESS from the Source.

Upon receiving the request from the Source Manager, the Source immediately prepares to terminate execution.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_SEQERROR     /* Operation invoked in invalid state */
```

See Also

- DG_CONTROL / DAT_IDENTITY / MSG_OPENDS

DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS

(from Source Manager to Source)

Call

```
DS_Entry(pOrigin, DG_CONTROL, DAT_IDENTITY, MSG_CLOSEDS, pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure

Valid States

4 only (Transitions Source back to the "loaded but not open" State - approximately State 3.5)

Description

Closes the Source so it can be unloaded from memory. The Source responds by doing its shutdown and clean-up activities needed to ensure the heap will be "clean" after the Source is unloaded. Under Windows, the Source will only be unloaded if the connection with the last application accessing it is about to be broken. The Source will know this by its internal "connect count" that should be maintained by any Source that supports multiple application connects.

Source Manager

pSourceIdentity is filled from a previous MSG_OPENDS operation.

Source

Perform all necessary housekeeping in anticipation of being unloaded. Be sure to dispose of any memory buffers that the Source has allocated locally, or that may have become the Source's responsibility during the course of the TWAIN session. The Source exists in a shared memory environment. It is therefore critical that all remnants of the Source, except the entry point (initial) code, be removed as the Source prepares to be unloaded.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_OPERATIONERROR   /* Internal Source error; */
                         /* handled by the Source  */
```

See Also

- DG_CONTROL / DAT_IDENTITY / MSG_OPENDS

DG_CONTROL / DAT_IDENTITY / MSG_GET

(from Source Manager to Source)

Call

```
DS_Entry(pOrigin, DG_CONTROL, DAT_IDENTITY, MSG_GET, pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure

Valid States

3 through 7 (Yes, the Source must be able to return the identity before it is opened.)

Description

This operation triplet is generated only by the Source Manager and is sent to the Source. It returns the identity structure for the Source.

Source Manager

No special set up or action required.

Source

Fills in all fields of pSourceIdentity except the Id field which is only modified by the Source Manager. This structure was allocated by either the application or the Source Manager depending on which one initiated the MSG_OPENDS operation for the Source.

Note: Sources should locate the code that handles initialization of the Source (responding to MSG_OPENDS) and identification (DAT_IDENTITY / MSG_GET) in the segment first loaded when the DLL/code resource is invoked. Responding to the identification operation should not cause any other segments to be loaded. Code to handle all other operations and to support the user interface should be located in code segments that will be loaded upon demand. Remember, the Source is a "guest" of the application and needs to be sensitive to use of available memory and other system resources. The Source Manager's perceived performance may be adversely affected unless the Source efficiently handles identification requests.

Return Codes

```
TWRC_SUCCESS       /* This operation must succeed. */
```

See Also

DG_CONTROL / DAT_IDENTITY / MSG_GETDEFAULT

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_IDENTITY, MSG_GETDEFAULT,
pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure

Valid States

3 through 7

Description

Gets the identification information of the system default Source.

Application

No special set up or action required.

Source Manager

Fills the structure pointed to by pSourceIdentity with identifying information about the system default Source.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_NODS          /* no Sources found matching       */
                      /* application's SupportedGroups   */
   TWCC_LOWMEMORY     /* not enough memory to perform    */
                      /* this operation                  */
```

See Also

- DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST
  DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT
  DG_CONTROL / DAT_IDENTITY / MSG_OPENDS
  DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT

DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_IDENTITY, MSG_GETFIRST,
pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure

Valid States

3 through 7

Description

The application may obtain a list of all Sources that are currently available on the system which match the application's supported groups (DGs, that the application specified in the SupportedGroups field of its TW_IDENTITY structure). To obtain the complete list of all available Sources requires invocation of a series of operations. The first operation uses MSG_GETFIRST to find the first Source on "the list" (whichever Source the Source Manager finds first). All the following operations use DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT to get the identity information, one at a time, of all remaining Sources.

Note: The application must invoke the MSG_GETFIRST operation before a MSG_GETNEXT operation. If the MSG_GETNEXT is invoked first, the Source Manager will fail the operation (TWRC_ENDOFLIST).

If the application wants to cause a specific Source to be opened, one whose ProductName the application knows, it must first establish the existence of the Source using the MSG_GETFIRST/MSG_GETNEXT operations. Once the application has verified that the Source is available, it can request that the Source Manager open the Source using DG_CONTROL / DAT_IDENTITY / MSG_OPENDS. The application must not execute this operation without first verifying the existence of the Source because the results may be unpredictable.

Application

No special set up or action required.

Source Manager

Fills the TW_IDENTITY structure pointed to by pSourceIdentity with the identity information of the first Source found by the Source Manager within the TWAIN directory/folder.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_NODS           /* No Sources can be found      */
   TWCC_LOWMEMORY      /* Not enough memory to perform */
                       /* this operation               */
```

See Also

- DG_CONTROL / DAT_IDENTITY / MSG_GETDEFAULT
  DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT
  DG_CONTROL / DAT_IDENTITY / MSG_OPENDS
  DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT

DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_IDENTITY, MSG_GETNEXT,
pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure

Valid States

3 through 7

Description

The application may obtain a list of all Sources that are currently available on the system which match the application's supported groups (DGs, that the application specified in the SupportedGroups field of its TW_IDENTITY structure). To obtain the complete list of all available Sources requires invocation of a series of operations. The first operation uses DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST to find the first Source on "the list" (whichever Source the Source Manager finds first). All the following operations use MSG_GETNEXT to get the identity information, one at a time, of all remaining Sources.

Note: The application must invoke the MSG_GETFIRST operation before a MSG_GETNEXT operation. If the MSG_GETNEXT is invoked first, the Source Manager will fail the operation (TWRC_ENDOFLIST).

If the application wants to cause a specific Source to be opened, one whose ProductName the application knows, it must first establish the existence of the Source using the MSG_GETFIRST/MSG_GETNEXT operations. Once the application has verified that the Source is available, it can request that the Source Manager open the Source using DG_CONTROL / DAT_IDENTITY / MSG_OPENDS. The application must not execute this operation without first verifying the existence of the Source because the results may be unpredictable.

Application

No special set up or action required.

Source Manager

Fills the TW_IDENTITY structure pointed to by pSourceIdentity with the identity information of the next Source found by the Source Manager within the TWAIN directory/folder.

Return Codes

```
TWRC_SUCCESS
TWRC_ENDOFLIST        /* after MSG_GETNEXT if no more */
                      /* Sources                       */
TWRC_FAILURE
   TWCC_LOWMEMORY     /* not enough memory to perform */
                      /* this operation               */
```

See Also

- DG_CONTROL / DAT_IDENTITY / MSG_GETDEFAULT
  DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST
  DG_CONTROL / DAT_IDENTITY / MSG_OPENDS
  DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT

DG_CONTROL / DAT_IDENTITY / MSG_OPENDS *(from Application to Source Manager)*

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_IDENTITY, MSG_OPENDS,
pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure

Valid States

3 only (Transitions to State 4, if successful)

Description

Loads the specified Source into main memory and causes its initialization.

Application

The application may specify any available Source's TW_IDENTITY structure in pSourceIdentity. That structure may have been obtained using a MSG_GETFIRST, MSG_GETNEXT, or MSG_USERSELECT operation. If the session with the Source Manager was closed since the identity structure being used was obtained, the application must set the Id field to 0. This will cause the Source Manager to issue the Source a new Id. The application can have the Source Manager open the default Source by setting the ProductName field to "\0" (Null string) and the Id field to zero.

Source Manager

Opens the Source specified by pSourceIdentity and creates a unique Id value for this Source (under MS Windows, this assumes that the Source hadn't already been opened by another application). This value is recorded in pSourceIdentity->Id. The Source Manager passes the triplet on to the Source to have the remaining fields in pSourceIdentity filled in.

Upon receiving the request from the Source Manager, the Source fills in all the fields in pSourceIdentity except for Id. If an application tries to connect to a Source that is already connected to its maximum number of applications, the Source returns TWRC_FAILURE/TWCC_MAXCONNECTIONS.

Warning: The Source and application **must** not assume that the value written into pSourceIdentity.Id will remain constant between sessions. This value is used internally by the Source Manager to uniquely identify applications and Sources and to manage the connections between them. During a different session, this value may still be valid but might be assigned to a different application or Source! Don't use this value directly.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_LOWMEMORY         /* not enough memory to */
                          /* open the Source      */
   TWCC_MAXCONNECTIONS    /* Source cannot support*/
                          /* another connection   */
   TWCC_NODS              /* specified Source was */
                          /* not found            */
   TWCC_OPERATIONERROR    /* internal Source error;*/
                          /* handled by the Source */
```

See Also

- DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS
  DG_CONTROL / DAT_IDENTITY / MSG_GET
  DG_CONTROL / DAT_IDENTITY / MSG_GETDEFAULT
  DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST
  DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT
  DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT

DG_CONTROL / DAT_IDENTITY / MSG_OPENDS *(from Source Manager to Source)*

Call

```
DS_Entry(pOrigin, DG_CONTROL, DAT_IDENTITY, MSG_OPENDS, pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure

Valid States

Source is loaded but not yet open (approximately State 3.5, session transitions to State 4, if successful)

Description

Opens the Source for operation.

Source Manager

pSourceIdentity is filled in from a previous DG_CONTROL / DAT_IDENTITY / MSG_GET and the Id field should be filled in by the Source Manager.

Source

Initializes any needed internal structures, performs necessary checks, and loads all resources needed for normal operation.

MS Windows only: Source should record a copy of *pOrigin, the application's TW_IDENTITY structure, whose Id field maintains a unique number identifying the application that is calling. Sources that support only a single connection should examine pOrigin->Id for each operation to verify they are being called by the application they acknowledge being connected with. All requests from other applications should fail (TWRC_FAILURE / TWCC_MAXCONNECTIONS). The Source is responsible for managing this, not the Source Manager (the Source Manager does not know in advance how many connections the Source will support).

Macintosh Note: Since the Source(s) and the Source Manager connected to a particular application live within that application's heap space, and are not shared with any other application, the discussion about multiply-connected Sources and verifying which application is invoking an operation is not relevant. A Source or Source Manager on the Macintosh can only be connected to a single application, though multiple copies of a Source or the Source Manager may be active on the same host simultaneously. These instances simply exist in different applications' heaps. If the instances need to communicate with one another, they might use a special resource file or may use the program's resource file on disk. The Source Manager manages its various instances in this way.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_LOWMEMORY          /* not enough memory to  */
                           /* open the Source       */
   TWCC_MAXCONNECTIONS     /* Source cannot support */
                           /* another connection    */
   TWCC_OPERATIONERROR     /* internal Source error;*/
                           /* handled by the Source */
```

See Also

- DG_CONTROL / DAT_IDENTITY / MSG_CLOSEDS
  DG_CONTROL / DAT_IDENTITY / MSG_GET

DG_CONTROL / DAT_IDENTITY / MSG_USERSELECT

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_IDENTITY, MSG_USERSELECT,
pSourceIdentity);
```

pSourceIdentity = A pointer to a TW_IDENTITY structure

Valid States

3 through 7

Description

This operation should be invoked when the user chooses **Select Source...** from the application's **File** menu (or an equivalent user action). This operation causes the Source Manager to display the Select Source dialog. This dialog allows the user to pick which Source will be used during subsequent **Acquire** operations. The Source selected becomes the system default Source. This default persists until a different Source is selected by the user. The system default Source may be overridden by an application (the override is local to only that application). Only Sources that can supply data matching one or more of the application's SupportedGroups (from the application's identity structure) will be selectable. All others will be unavailable for selection.

Application

If the application wants a particular Source, other than the system default, to be highlighted in the Select Source dialog, it should set the ProductName field of the structure pointed to by pSourceIdentity to the ProductName of that Source. This information should have been obtained from an earlier operation using DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST, MSG_GETNEXT, or MSG_USERSELECT. Otherwise, the application should set the ProductName field in pSourceIdentity to the null string ("\0"). In either case, the application should set the Id field in pSourceIdentity to zero.

If the Source Manager can't find a Source whose ProductName matches that specified by the application, it will select the system default Source (the default that matches the SupportedGroups of the application). This is not considered to be an error condition. No error will be reported. The application should check the ProductName field of pSourceIdentity following this operation to verify that the Source it wanted was opened.

Source Manager

The Source Manager displays the Select Source dialog and allows the user to select a Source. When the user clicks the "OK" button ("Select" button in the MS Windows Source Manager) in the Select Source dialog, the system default Source (maintained by the Source Manager) will be changed to the selected Source. This Source's identifying information will be written into pSourceIdentity.

The "Select" button ("OK" button in the Macintosh Source Manager) will be grayed out if there are no Sources available matching the SupportedGroups specified in the application's identity structure, pOrigin. The user must click the "Cancel" button to exit the Select Source dialog. The application cannot discern from this Return Code whether the user simply canceled the selection or there were no Sources for the user to select. If the application really wants to know whether any Sources are available that match the specified SupportedGroups it can invoke a MSG_GETFIRST operation and check for a successful result.

It copies the TW_IDENTITY structure of the selected Source into pSourceIdentity.

Suggestion for Source Developers: The string written in the Source's TW_IDENTITY.ProductName field should clearly and unambiguously identify your product or the Source to the user (if the Source can be used to control more than one device). ProductName contains the string that will be placed in the Select Source dialog (accompanied, on the Macintosh, with an icon from the Source's resource file representing the Source). It is further suggested that the Source's disk file name approximate the ProductName to assist the user in equating the two.

Return Codes

```
TWRC_SUCCESS
TWRC_CANCEL          /* User clicked cancel button - maybe there   */
                     /* were no Sources                            */
TWRC_FAILURE
   TWCC_LOWMEMORY    /* not enough memory to perform this  */
                     /* operation                          */
```

See Also

- DG_CONTROL / DAT_IDENTITY / MSG_GETDEFAULT
  DG_CONTROL / DAT_IDENTITY / MSG_GETFIRST
  DG_CONTROL / DAT_IDENTITY / MSG_GETNEXT
  DG_CONTROL / DAT_IDENTITY / MSG_OPENDS

DG_CONTROL / DAT_NULL / MSG_CLOSEDSREQ *(from Source to Application - Windows only)*

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_NULL, MSG_CLOSEDSREQ, NULL);
```

This operation requires no data (NULL)

Valid States

5 through 7 (This operation causes the session to transition to State 5.)

Description

While the Source is enabled, the application is sending all events/messages to the Source. The Source will use one of these events/messages to indicate to the application that it needs to be closed.

On Windows, the Source sends this DG_CONTROL / DAT_NULL / MSG_CLOSEDSREQ to the Source Manager to cause the Source Manager to post a private message to the application's event/message loop. This guarantees that the application will have an event/message to pass to the Source Manager so it will be able to communicate the Source's Close request back to the application.

On Macintosh, the application simply sends Null events to the Source periodically to ensure it has a communication carrier when needed. Therefore, this operation is not used on a Macintosh implementation.

Source (on Windows only)

Source creates this triplet with NULL data and sends it to the Source Manager via the Source Manager's DSM_Entry point.

pDest is the TW_IDENTITY structure of the application.

Source Manager (on Windows only)

Upon receiving this triplet, the Source Manager posts a private message to the application's event/message loop. Since the application is forwarding all events/messages to the Source while the Source is enabled, this creates a communication device needed by the Source. When this private message is received by the Source Manager (via the DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation), the Source Manager will insert a MSG_CLOSEDSREQ into the TWMessage field on behalf of the Source.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_SEQERROR    /* Operation invoked in invalid state */
   TWCC_BADDEST     /* No such application in session with*/
                    /* Source                             */
```

See Also

- DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT
  DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS

DG_CONTROL / DAT_NULL / MSG_XFERREADY *(from Source to Application - applies to Windows only)*

Call:

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_NULL, MSG_XFERREADY, NULL);
```

This operation requires no data (NULL)

Valid States

State 5 only (This operation causes the transition to State 6.)

Description

While the Source is enabled, the application is sending all events/messages to the Source. The Source will use one of these events/ messages to indicate to the application that the data is ready to be transferred.

On Windows, the Source sends this DG_CONTROL / DAT_NULL / MSG_XFERREADY to the Source Manager to cause the Source Manager to post a private message to the application's event/message loop. This guarantees that the application will have an event/message to pass to the Source and the Source will be able to communicate its "transfer ready" announcement back to the application.

On Macintosh, the application simply sends Null events to the Source periodically to ensure it has a communication carrier when needed. Therefore, this operation is not used on a Macintosh implementation.

Source (on Windows only)

Source creates this triplet with NULL data and sends it to the Source Manager via the Source Manager's DSM_Entry point.

pDest is the TW_IDENTITY structure of the application.

Source Manager

Upon receiving this triplet, the Source Manager posts a private message to the application's event/message loop. Since the application is forwarding all events/messages to the Source while the Source is enabled, this creates a communication device needed by the Source. When this private message is received by the Source Manager (via the DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT operation), the Source Manager will insert the MSG_XFERREADY into the TWMessage field on behalf of the Source.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_SEQERROR     /* Operation invoked in invalid state */
   TWCC_BADDEST      /* No such application in session with*/
                     /* Source                             */
```

See Also

- DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT
  DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET
  DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET
  DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET

DG_CONTROL / DAT_PARENT / MSG_CLOSEDSM

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_PARENT, MSG_CLOSEDSM, pParent);
```

On Windows

pParent = points to the window handle (hWnd) that will act as the Source's "parent". The variable is of type TW_INT32 and the low word of this variable must contain the window handle.

On Macintosh

pParent = should be a 32-bit NULL value

Valid States

3 **only** (causes transition back to State 2, if successful)

Description

When the application has closed all the Sources it had previously opened, and is finished with the Source Manager (the application plans to initiate no other TWAIN sessions), it must close the Source Manager. The application should unload the Source Manager DLL or code resource after the Source Manager is closed—unless the application has immediate plans to use the Source Manager again.

Application

References the same pParent parameter that was used during the "open Source Manager" operation. If the operation returns TWRC_SUCCESS, the application should unload the Source Manager from memory.

Source Manager

Does any housekeeping needed to prepare for being unloaded from memory. This housekeeping is transparent to the application.

MS Windows only—If the Source Manager is open to at least one other application, it will clean up just activities relative to the closing application, then return TWRC_SUCCESS. The application will attempt to unload the Source Manager DLL. Windows will tell the application that the unload was successful, but the Source Manager will remain active and connected to the other application(s).

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_SEQERROR      /* Operation invoked in invalid state */
```

See Also

- DG_CONTROL / DAT_PARENT / MSG_OPENDSM

DG_CONTROL / DAT_PARENT / MSG_OPENDSM

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_PARENT, MSG_OPENDSM, pParent);
```

On Windows

pParent = points to the window handle (hWnd) that will act as the Source's "parent". The variable is of type TW_INT32 and the low word of this variable must contain the window handle

On Macintosh

pParent = should be a 32-bit NULL value

Valid States

2 **only** (causes transition to State 3, if successful)

Description

Causes the Source Manager to initialize itself. This operation **must** be executed before any other operations will be accepted by the Source Manager.

Application

MS Windows only—The application should set the pParent parameter to point to a window handle (hWnd) of an open window that will remain open until the Source Manager is closed. If application can't open the Source Manager DLL, Windows displays an error box (this error box can be disabled by a prior call to SetErrorMode (SET_NOOPENFILEERRORBOX)).

Macintosh only—Set pParent to NULL.

Source Manager

Initializes and prepares itself for subsequent operations. Maintains a copy of pParent.

MS Windows only—If Source Manager is already open, Source Manager won't reinitialize but will retain a copy of pParent.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_LOWMEMORY      /* not enough memory to perform */
                       /* this operation               */
   TWCC_SEQERROR       /* Operation invoked in invalid */
                       /* state                        */
```

See Also

- DG_CONTROL / DAT_PARENT / MSG_CLOSEDSM

DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_PENDINGXERS, MSG_ENDXFER,
pPendingXfers);
```

pPendingXfers = A pointer to a TW_PENDINGXFERS structure

Valid States

6 and 7

(Transitions to State 5 if this was the last transfer (pPendingXfers->Count == 0). Transitions to State 6 if there are more transfers pending (pPendingXfers->Count != 0). To abort all remaining transfers and transition from State 6 to State 5, use DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET.

Description

This triplet is used to cancel or terminate a transfer. Issued in State 6, this triplet cancels the next pending transfer, discards the transfer data, and decrements the pending transfers count. In State 7, this triplet terminates the current transfer. If any data has not been transferred (this is only possible during a memory transfer) that data is discarded.

The application can use this operation to cancel the next pending transfer (**Source writers take note of this**). For example, after the application checks TW_IMAGEINFO, it may decide to not transfer the next image. The operation must be sent prior to the beginning of the transfer, otherwise the Source will simply abort the current transfer. The Source decrements the number of pending transfers.

Application

The application must invoke this operation at the end of every transfer to signal the Source that the application has received all the data it expected. The application should send this after receiving a TWRC_XFERDONE or TWRC_CANCEL.

No special set up or action required. Be sure to correctly track which state the Source will be in as a result of your action. Be aware of the value in pPendingXfers->Count both before and after the operation. Invoking this operation causes the loss of data that your user may not expect to be lost. Be very careful and prudent when using this operation.

Source

Option #1) Fill pPendingXfers->Count with the number of transfers the Source is ready to supply to the application, upon demand. If pPendingXfers->Count > 0 (or equals -1), transition to State 6 and await initiation of the next transfer by the application. If pPendingXfers->Count == 0, transition all the way back to State 5 and await the next acquisition.

Option #2) Preempt the acquired data that is next in line for transfer to the application (pending transfers can be thought of as being pushed onto a FIFO queue as acquired and popped off the queue when transferred). Decrement pPendingXfers->Count. If already acquired, discard the data for the preempted transfer. Update pPendingXfers->Count with the new number of pending transfers. If this value is indeterminate, leave the value in this field at -1.

Option #3) Cancel the current transfer. Discard any local buffers or data involved in the transfer. Prepare the Source and the device for the next transfer. Decrement pPendingXfers->Count (don't decrement if already zero or -1). If there is a transfer pending, return to State 6 and prepare the Source to begin the next transfer. If no transfer is pending, return to State 5 and await initiation of the next acquisition from the application or the user.

Note: If a Source supports simultaneous connections to more than one application, the Source should maintain a separate pPendingXfers structure for each application it is in-session with.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST     /* No such Source in-session with */
                    /* application                    */
   TWCC_SEQERROR    /* Operation invoked in invalid   */
                    /* state                          */
```

See Also

- DG_CONTROL / DAT_IMAGEFILEXFER / MSG_GET
  DG_CONTROL / DAT_IMAGEMEMXFER / MSG_GET
  DG_CONTROL / DAT_IMAGENATIVEXFER / MSG_GET
  DG_CONTROL / DAT_PENDINGXFERS / MSG_GET
  DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET
- Capability - CAP_XFERCOUNT

DG_CONTROL / DAT_PENDINGXFERS / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_PENDINGXERS, MSG_GET,
pPendingXfers);
```

pPendingXfers = A pointer to a TW_PENDINGXFERS structure

Valid States

4 through 7

Description

Returns the number of transfers the Source is ready to supply to the application, upon demand.

Application

No special set up or action required.

Source

Fill pPendingXfers->Count with the number of transfers the Source is ready to supply to the application, upon demand. This value should reflect the number of complete data blocks that have already been acquired or are in the process of being acquired.

If the Source is not sure how many transfers are pending, but is sure that the number is at least one, set pPendingXfers->Count to -1. A Source connected to a device with an automatic document feeder that cannot determine the number of pages in the feeder, or how many selections the user may make on each page, would respond in this way. A Source providing access to a series of images from a video camera or a data base may also respond this way.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST     /* No such Source in-session with */
                    /* application                    */
   TWCC_SEQERROR    /* Operation invoked in invalid   */
                    /* state                          */
```

See Also

- DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER
  DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET
- Capability - CAP_XFERCOUNT

DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_PENDINGXERS, MSG_RESET,
pPendingXfers);
```

pPendingXfers = A pointer to a TW_PENDINGXFERS structure.

Valid States

6 only (Transitions to State 5, if successful)

Description

Sets the number of pending transfers in the Source to zero.

Application

No special set up or action required. Be aware of the state transition caused by this operation. Invoking this operation causes the loss of data that your user may not expect to be lost. Be very careful and prudent when using this operation.

The application may need to use this operation if an error occurs within the application that necessitates breaking off all TWAIN sessions. This will get the application, Source Manager, and Source back to State 5 together.

Source

Set pPendingXfers->Count to zero. Discard any local buffers or data involved in any of the pending transfers. Return to State 5 and await initiation of the next acquisition from the application or the user.

Note: If a Source supports simultaneous sessions with more than one application, the Source should maintain a separate pPendingXfers structure for each application it is in-session with.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST     /* No such Source in-session with */
                    /* application                    */
   TWCC_SEQERROR    /* Operation invoked in invalid   */
                    /* state                          */
```

See Also

- DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER
  DG_CONTROL / DAT_PENDINGXFERS / MSG_GET
- Capability - CAP_XFERCOUNT

DG_CONTROL / DAT_SETUPFILEXFER / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_SETUPFILEXFER, MSG_GET, pSetupFile);
```

pSetupFile = A pointer to a TW_SETUPFILEXFER structure

Valid States

4 through 6

Description

Returns information about the file into which the Source has or will put the acquired data.

Application

No special set up or action required.

Source

Set the following:

pSetupFile->Format = format of destination file
(Constants are TWFF_TIFF, TWFF_PICT, TWFF_BMP, etc.)

pSetupFile->FileName = name of file (on Windows, include the complete path name)

pSetupFile->VRefNum = volume reference number
(**Macintosh only**)

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST       /* No such Source in-session    */
                      /* with application             */
   TWCC_BADPROTOCOL   /* Source does not support      */
                      /* file transfer                */
   TWCC_SEQERROR      /* Operation invoked in invalid */
                      /* state                        */
```

See Also

- DG_CONTROL / DAT_SETUPFILEXFER / MSG_GETDEFAULT
  DG_CONTROL / DAT_SETUPFILEXFER / MSG_RESET
  DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET
  DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET
- Capabilities - ICAP_XFERMECH, ICAP_IMAGEFILEFORMAT

DG_CONTROL / DAT_SETUPFILEXFER / MSG_GETDEFAULT

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_SETUPFILEXFER, MSG_GETDEFAULT,
pSetupFile);
```

pSetupFile = A pointer to a TW_SETUPFILEXFER structure

Valid States

4 through 6

Description

Returns the file information for the default file

Macintosh only—VRefNum should be set only if the default file exists. Otherwise, set this field to NULL.

Application

No special set up or action required.

Source

Set the following:

pSetupFile->Format = format of destination file
(Constants are TWFF_TIFF, TWFF_PICT, TWFF_BMP, etc.)

pSetupFile->FileName = name of file (on Windows, include the complete path name)

pSetupFile->VRefNum = volume reference number
(**Macintosh only**)

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST      /* No such Source in-session    */
                     /* with application             */
   TWCC_BADPROTOCOL  /* Source does not support      */
                     /* file transfer                */
   TWCC_SEQERROR     /* Operation invoked in invalid */
                     /* state                        */
```

See Also

- DG_CONTROL / DAT_SETUPFILEXFER / MSG_GET
  DG_CONTROL / DAT_SETUPFILEXFER / MSG_RESET
  DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET
  DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET
- Capabilities - ICAP_XFERMECH, ICAP_IMAGEFILEFORMAT

DG_CONTROL / DAT_SETUPFILEXFER / MSG_RESET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_SETUPFILEXFER, MSG_RESET,
pSetupFile);
```

pSetupFile = A pointer to a TW_SETUPFILEXFER structure

Valid States

4 only

Description

Resets the current file information to the default file information and returns that default information.

Application

No special set up or action required.

Source

Set the following to reflect the default file used by the Source:

pSetupFile->Format = format of destination file
(Constants are TWFF_TIFF, TWFF_PICT, TWFF_BMP, etc.)

pSetupFile->FileName = name of file (on Windows, include the complete path name)

pSetupFile->VRefNum = volume reference number
(**Macintosh only**)

VRefNum should be set to reflect the default file only if it already exists. Otherwise, set this field to NULL.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST       /* No such Source in-session with*/
                      /* application                   */
   TWCC_BADPROTOCOL   /* Source does not support file  */
                      /* transfer                      */
   TWCC_SEQERROR      /* Operation invoked in invalid  */
                      /* state                         */
```

See Also

- DG_CONTROL / DAT_SETUPFILEXFER / MSG_GET
  DG_CONTROL / DAT_SETUPFILEXFER / MSG_GETDEFAULT
  DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET
  DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET

- Capabilities - ICAP_XFERMECH, ICAP_IMAGEFILEFORMAT

DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_SETUPFILEXFER, MSG_SET, pSetupFile);
```

pSetupFile = A pointer to a TW_SETUPFILEXFER structure

Valid States

4 through 6

Description

Sets the file transfer information for the next file transfer. The application is responsible for verifying that the specified file name is valid and that the file either does not currently exist (in which case, the Source is to create the file), or that the existing file is available for opening and read/write operations.

The application should also assure that the file format it is requesting can be provided by the Source (otherwise, the Source will generate a TWRC_FAILURE/TWCC_BADVALUE error).

Application

Set the following:

pSetupFile->Format = format of destination file
(Constants are TWFF_TIFF, TWFF_PICT, TWFF_BMP, etc.)

pSetupFile->FileName = name of file (on Windows, include the complete path name)

pSetupFile->VRefNum = volume reference number
(**Macintosh only**)

Note: ICAP_XFERMECH must have been set to TWSX_FILE during previous capability negation.

Source

Use the specified file format and file name information to transfer the next file to the application. If any part of the information being set is wrong or missing, use the Source's default file (TWAIN.TMP in the current directory) and return TWRC_FAILURE with TWCC_BADVALUE. If the format and file name are OK, but a file error occurs when trying to open the file (other than "file does not exist"), return TWCC_BADVALUE and set up to use the default file. If the specified file does not exit, create it. If the file exists and has data in it, overwrite the existing data starting with the first byte of the file.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST        /* No such Source in-session    */
                       /* with application             */
   TWCC_BADPROTOCOL    /* Source doesn't support       */
                       /* file transfer                */
   TWCC_BADVALUE       /* Source cannot comply with    */
                       /* one of the settings          */
   TWCC_SEQERROR       /* Operation invoked in invalid */
                       /* state                        */
```

See Also

- DG_CONTROL / DAT_SETUPFILEXFER / MSG_GET
  DG_CONTROL / DAT_SETUPFILEXFER / MSG_GETDEFAULT
  DG_CONTROL / DAT_SETUPFILEXFER / MSG_RESET
  DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET
- Capabilities - ICAP_XFERMECH, ICAP_IMAGEFILEFORMAT

DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_SETUPMEMXFER, MSG_GET, pSetupMem);
```

pSetupMem = A pointer to a TW_SETUPMEMXFER structure

Valid States

4 through 6

Description

Returns the Source's preferred, minimum, and maximum allocation sizes for transfer memory buffers. The application using buffered memory transfers must use a buffer size between MinBufSize and MaxBufSize in their TW_IMAGEMEMXFER.Memory.Length when using the DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET operation. Sources may return a more efficient preferred value in State 6 after the image size, etc. has been specified.

Application

No special set up or action required.

Source

Set the following:

pSetupMem->MinBufSize = minimum usable buffer size, in bytes

pSetupMem->MaxBufSize = maximum usable buffer size, in bytes (-1 means an indeterminately large buffer is acceptable)

pSetupMem->Preferred = preferred transfer buffer size, in bytes

If the Source doesn't care about the size of any of these specifications, set the field(s) to TWON_DONTCARE32. This signals the application that any value for that field is OK with the Source.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST     /* No such Source in-session with */
                    /* application                    */
   TWCC_SEQERROR    /* Operation invoked in invalid   */
                    /* state                          */
```

See Also

- DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET
- Capability - ICAP_COMPRESSION, ICAP_XFERMECH

DG_CONTROL / DAT_STATUS / MSG_GET

(from Application to Source Manager)

Call

```
DSM_Entry(pOrigin, NULL, DG_CONTROL, DAT_STATUS, MSG_GET, pSourceStatus);
```

pSourceStatus = A pointer to a TW_STATUS structure

Valid States

2 through 7

Description

Returns the current Condition Code for the Source Manager.

Application

NULL references the operation to the Source Manager.

Source Manager

Fills pSourceStatus->ConditionCode with its current Condition Code. Then, it will clear its internal Condition Code so you cannot issue a status inquiry twice for the same error (the information is lost after the first request).

Return Codes

```
TWRC_SUCCESS       /* This operation must succeed     */
TWRC_FAILURE
   TWCC_BADDEST    /* No such Source in-session with */
                   /* application                    */
```

See Also

- Return Codes and Condition Codes (Chapter 10)

DG_CONTROL / DAT_STATUS / MSG_GET *(from Application to Source)*

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_STATUS, MSG_GET, pSourceStatus);
```

pSourceStatus = A pointer to a TW_STATUS structure

Valid States

4 through 7

Description

Returns the current Condition Code for the specified Source.

Application

pDest references a copy of the targeted Source's identity structure.

Source

Fills pSourceStatus->ConditionCode with its current Condition Code. Then, it will clear its internal Condition Code so you cannot issue a status inquiry twice for the same error (the information is lost after the first request).

Return Codes

```
TWRC_SUCCESS        /* This operation must succeed     */
TWRC_FAILURE
   TWCC_BADDEST     /* No such Source in-session with */
                    /* application                    */
```

See Also

- Return Codes and Condition Codes (Chapter 10)

DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_USERINTERFACE, MSG_DISABLEDS,
pUserInterface);
```

pUserInterface = A pointer to a TW_USERINTERFACE structure

Valid States

5 only (Transitions to State 4, if successful)

Description

This operation causes the Source's user interface, if displayed during the DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS operation, to be lowered. The Source is returned to State 4, where capability negotiation can again occur. The application can invoke this operation either because it wants to shut down the current session, or in response to the Source "posting" a MSG_CLOSEDSREQ event to it. Rarely, the application may need to close the Source because an error condition was detected.

Application

References the same pUserInterface structure as during the MSG_ENABLEDS operation. This implies that the application keep a copy of this structure locally as long as the Source is enabled.

If the application did not display the Source's built-in user interface, it will most likely invoke this operation either when all transfers have been completed or aborted (TW_PENDINGXFERS.Count = 0).

Source

If the Source's user interface is displayed, it should be lowered. The Source returns to State 4 and is again available for capability negotiation.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST       /* No such Source in-session */
                      /* with application          */
   TWCC_SEQERROR      /* Operation invoked in      */
                      /* invalid state             */
```

See Also

- DG_CONTROL / DAT_NULL / MSG_CLOSEDSREQ
  DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS
- Event loop information (in Chapter 3)

DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_USERINTERFACE, MSG_ENABLEDS,
pUserInterface);
```

pUserInterface = A pointer to a TW_USERINTERFACE structure

Valid States

4 only (Transitions to State 5, if successful)

Description

This operation causes three responses in the Source:

- Places the Source into a "ready to acquire" condition. If the application raises the Source's user interface (see #2, below), the Source will wait to assert MSG_XFERREADY until the "GO" button in its user interface or on the device is clicked. If the application bypasses the Source's user interface, this operation causes the Source to become immediately "armed". That is, the Source should assert MSG_XFERREADY as soon as it has data to transfer.
- The application can choose to raise the Source's built-in user interface, or not, using this operation. The application signals the Source's user interface should be displayed by setting pUserInterface->ShowUI to TRUE. If the application does not want the Source's user interface to be displayed, or wants to replace the Source's user interface with one of its own, it sets pUserInterface->ShowUI to FALSE. If activated, the Source's user interface will remain displayed until it is closed by the user or explicitly disabled by the application (see Note).
- Terminates Source's acceptance of "set capability" requests from the application. Capabilities can only be negotiated in State 4 (unless special arrangements are made using the CAP_EXTENDEDCAPS capability). Values of capabilities can still be inquired in States 5 through 7.

Note: Once the Source is enabled, the application **must** begin sending the Source every event that enters the application's main event loop. The application must continue to send the Source events until it disables (MSG_DISABLEDS) the Source. This is true even if the application chooses not to use the Source's built-in user interface.

Application

Set pUserInterface->ShowUI to TRUE to display the Source's built-in user interface, or to FALSE to place the Source in an "armed" condition so that it is immediately prepared to acquire data for transfer. Set ShowUI to FALSE only if bypassing the Source's built-in user interface—that is, only if the application is prepared to handle all user interaction necessary to acquire data from the selected Source.

Sources are not required to be enabled without showing their User Interface (i.e. TW_USERINTERFACE.ShowUI = FALSE). If a Source does not support ShowUI = FALSE, they will continue to be enabled just as if ShowUI = TRUE, but return TWRC_CHECKSTATUS. The application can check for this Return Code and continue knowing the Source's User Interface is being displayed.

Watch the value of pUserInterface->ModalUI after the operation has completed to see if the Source's user interface is modal or modeless.

The application must maintain a local copy of pUserInterface while the Source is enabled.

MS Windows only—The application should place a handle (hWnd) to the window acting as the Source's parent into pUserInterface->hParent.

Macintosh only—Set pUserInterface->hParent to NULL.

Note: Application should establish that the Source can supply compatible ICAP_PIXELTYPEs and ICAP_BITDEPTHs prior to enabling the Source. The application **must** verify that the Source can supply data of a type it can consume. If this operation fails, the application should notify the user that the device and application are incompatible due to data type mismatch. If the application diligently sets SupportedGroups in its identity structure before it tries to open the Source, the Source Manager will, in the Select Source dialog or through the MSG_GETFIRST/MSG_GETNEXT mechanism, filter out the Sources that don't match these SupportedGroups.

Source

If pUserInterface->ShowUI is TRUE, the Source should display its user interface and wait for the user to initiate an acquisition. If pUserInterface->ShowUI is FALSE, the Source should immediately begin acquiring data based on its current configuration (a device that requires the user to push a button on the device, such as a hand-scanner, will be "armed" by this operation and will assert MSG_XFERREADY as soon as the Source has data ready for transfer). The Source should fail any attempt to set a capability value (TWRC_FAILURE / TWCC_SEQERROR) until it returns to State 4 (unless an exception approval exists via a CAP_EXTENDEDCAPS agreement).

Set pUserInterface->ModalUI to TRUE if your built-in user interface is modal. Otherwise, set it to FALSE.

Note: While the Source's user interface is raised, the Source is responsible for presenting the user with appropriate progress indicators regarding the acquisition and transfer processes unless the application has set CAP_INDICATORS to FALSE. The Source must also report errors to the user (without regard for the settings of CAP_INDICATORS and ShowUI, i.e. they may be set to FALSE and errors still must be reported).

It is strongly recommended that all Sources support being enabled without their User Interface if the application requests (TW_USERINTERFACE.ShowUI = FALSE). But if your Source cannot be used without its User Interface, it should enable showing the Source User Interface (just as if ShowUI = TRUE) but return TWRC_CHECKSTATUS. All Sources, however, must support the CAP_UICONTROLLABLE. This capability reports whether or not a Source allows ShowUI = FALSE. An application can use this capability to know whether the Source-supplied user interface can be suppressed before it is displayed.

Return Codes

```
TWRC_SUCCESS
TWRC_CHECKSTATUS            /* Source cannot enable     */
                            /* without User Interface   */
                            /* so it enabled with the   */
                            /* User Interface.          */
TWRC_FAILURE
   TWCC_BADDEST             /* No such Source in-session */
                            /* with application          */
   TWCC_LOWMEMORY           /* Not enough memory to open */
                            /* the Source                */
   TWCC_OPERATIONERROR      /* Internal Source error;    */
                            /* handled by the Source     */
   TWCC_SEQERROR            /* Operation invoked in      */
                            /* invalid state             */
```

See Also

- DG_CONTROL / DAT_NULL / MSG_CLOSEDSREQ
  DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS
- Capability - CAP_INDICATORS
- Event loop information (in Chapter 3)

DG_CONTROL / DAT_XFERGROUP / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_CONTROL, DAT_XFERGROUP, MSG_GET, pXferGroup);
```

pXferGroup = A pointer to a TW_UINT32 value

Valid States

4 through 6

Description

Returns the Data Group (the type of data) for the upcoming transfer. The Source is required to only supply one of the DGs specified in the SupportedGroups field of pOrigin.

Application

Should have previously (during a DG_CONTROL / DAT_PARENT / MSG_OPENDSM) set pOrigin. SupportedGroups to reflect the DGs the application is interested in receiving from a Source. Since DG_XXXX identifiers are bit flags, the application can perform a bitwise OR of DG_XXXXx constants of interest to build the SupportedGroups field (this is appropriate when more kinds of data than DG_IMAGE are available).

Note: Version 1.x of the Toolkit defines DG_IMAGE as the sole Data Group (DG_CONTROL is masked from any processing of SupportedGroups). Future versions of TWAIN may define support for other DGs.

Source

Set pXferGroup to the DG_XXXX constant that identifies the type of data that is ready for transfer from the Source (DG_IMAGE is the only non-custom Data Group defined in TWAIN version 1.x).

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST       /* No such Source in-session with */
                      /* application                    */
   TWCC_SEQERROR      /* Operation invoked in invalid   */
                      /* state                          */
```

See Also

DG_IMAGE / DAT_CIECOLOR / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_CIECOLOR,
          MSG_GET, pCIEColor);
```

pCIEColor = A pointer to a TW_CIECOLOR structure

Valid States

4 through 6

Description

Background - The DAT_CIECOLOR data argument type is used to communicate the parametrics for performing a transformation from any arbitrary set of tri-stimulus values into CIE XYZ color space. Color data stored in this format is more readily manipulated mathematically than some other spaces. See Appendix A for more information about the definitions and data structures used to describe CIE color data within TWAIN.

This operation causes the Source to report the currently active parameters to be used in converting acquired color data into CIE XYZ.

Application

Prior to invoking this operation, the application should establish that the Source can provide data in CIE XYZ form. This can be determined by invoking a MSG_GET on ICAP_PIXELTYPE. If TWPT_CIEXYZ is one of the supported types, then these operations are valid. The application can specify that transfers should use the CIE XYZ space by invoking a MSG_SET operation on ICAP_PIXELTYPE using a TW_ONEVALUE container structure whose value is TWPT_CIEXYZ.

No special set up is required. Invoking this operation following the transfer (after the Source is back in State 6) will guarantee that the exact parameters used to convert the image are reported.

Source

Fill pCIEColor with the current values applied in any conversion of image data to CIE XYZ. If no values have been set by the application, fill the structure with either the values calculated for this image or the Source's default values, whichever most accurately reflect the state of the Source.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL      /* Source does not support the  */
                         /* CIE descriptors              */
   TWCC_SEQERROR         /* Operation invoked in invalid */
                         /* state                        */
```

See Also

- Capability - ICAP_PIXELTYPE
- Appendix A

DG_IMAGE / DAT_GRAYRESPONSE / MSG_RESET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_GRAYRESPONSE, MSG_RESET, pResponse);
```

pResponse = A pointer to a TW_GRAYRESPONSE structure

Valid States

4 only

Description

Background - The two DAT_GRAYRESPONSE operations allow the application to specify a transfer curve that the Source should apply to the grayscale it acquires. This curve should be applied to the data prior to transfer. The Source should maintain an "identity response curve" to be used when it is MSG_RESET.

The MSG_RESET operation causes the Source to use its "identity response curve." The identity curve causes no change in the values of the captured data when it is applied.

Application

No special action.

Source

Apply the identity response curve to all future grayscale transfers. This means that the Source will transfer the grayscale data exactly as acquired.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL      /* Source does not support      */
                         /* grayscale response curves    */
   TWCC_SEQERROR         /* Operation invoked in invalid */
                         /* state                        */
```

See Also

- DG_IMAGE / DAT_GRAYRESPONSE / MSG_SET
- Capability - ICAP_PIXELTYPE

DG_IMAGE / DAT_GRAYRESPONSE / MSG_SET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_GRAYRESPONSE, MSG_SET, pResponse);
```

pResponse = A pointer to a TW_GRAYRESPONSE structure

Valid States

4 only

Description

Background - The two DAT_GRAYRESPONSE operations allow the application to specify a transfer curve that the Source should apply to the grayscale it acquires. This curve should be applied to the data prior to transfer. The Source should maintain an "identity response curve" to be used when it is MSG_RESET. This identity curve should cause no change in the values of the data it is applied to.

This operation causes the Source to transform any grayscale data according to the response curve specified.

Application

All three elements of the response curve for any given index should hold the same value (the curve is stored in a TW_ELEMENT8 which contains three "channels" of data). The Source may not support this operation. The application should be diligent to examine the return code from this operation.

Source

Apply the specified response curve to all future grayscale transfers. The transformation should be applied before the data is transferred.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL      /* Source does not support      */
                         /* grayscale response curves    */
   TWCC_SEQERROR         /* Operation invoked in invalid */
                         /* state                        */
```

See Also

- DG_IMAGE / DAT_GRAYRESPONSE / MSG_RESET
- Capability - ICAP_PIXELTYPE

DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGEFILEXFER, MSG_GET, NULL);
```

This operation acts on NULL data. File information can be set with the DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET operation.

Valid States

6 only (Transitions to State 7, if successful. Remains in State 7 until MSG_ENDXFER operation.)

Description

This operation is used to initiate the transfer of an image from the Source to the application via the disk-file transfer mechanism. It causes the transfer to begin.

Application

No special set up or action required. Application should have already invoked the DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET operation unless the Source's default transfer format and file name (typically, TWAIN.TMP) are acceptable to the application. The application need only invoke this operation once per image transferred.

Notes: If the application is planning to receive multiple images from the Source while using the Source's default file name, the application should plan to pause between transfers and copy the file just written. The Source will overwrite the file unless it is instructed to write to a different file.

Applications can specify a unique file for each transfer using DAT_SETUPFILEXFER / MSG_SET in State 6 or 5 (and 4, of course).

Source

Acquire the image data, format it, create any appropriate header information, and write everything into the file specified by the previous DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET operation, and close the file.

Handling Possible File Conditions:

- **If the application did not set conditions up using the DAT_SETUPFILEXFER / MSG_SET operation during this session**, use your own default file name, file format, and location for the created file.
- **If the specified file already exists,** overwrite the file in place.
- **If the specified file does not exist**, create the file.
- **If the specified file exists and cannot be accessed**, or a system error occurs while writing the file, report the error to the user and return TWRC_FAILURE with TWCC_OPERATIONERROR. Stay in State 6. The file contents are invalid. The image whose transfer failed is still a pending transfer so do not decrement TW_PENDINGXFERS.Count.
- **If the file is written successfully**, return TWRC_XFERDONE.
- **If the user cancels the transfer**, return TWRC_CANCEL.

Return Codes

```
TWRC_XFERDONE
TWRC_CANCEL
TWRC_FAILURE
   TWCC_BADDEST           /* No such Source in-session */
                          /* with application          */
   TWCC_OPERATIONERROR    /* Failure in the Source --  */
                          /* transfer invalid          */
   TWCC_SEQERROR          /* Operation invoked in      */
                          /* invalid state             */
```

See Also

- DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET
  DG_IMAGE / DAT_IMAGEINFO / MSG_GET
  DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET,
- Capabilities - ICAP_XFERMECH, ICAP_IMAGEFILEFORMAT

DG_IMAGE / DAT_IMAGEINFO / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGEINFO, MSG_GET, pImageInfo);
```

pImageInfo = A pointer to a TW_IMAGEINFO structure

Valid States

6 only

Description

This operation provides to the application general image description information about the image about to be transferred. The same data structure type is used regardless of the mechanism used to transfer the image (Native, Disk File, or Buffered Memory transfer).

Application

The application can use this operation to check the parameters of the image before initiating the transfer. Applications may inform Sources that they accept -1 value for ImageHeight/ImageWidth by setting the ICAP_UNDEFINEDIMAGESIZE capability to TRUE. Note that the speed at which the application supplies buffers may determine the scanning speed.

Source

Fills in all fields in pImageInfo. All fields are filled in as you would expect with the following exceptions:

XResolution or YResolution

Set to -1 if the device creates data with no inherent resolution (such as a digital camera).

ImageWidth

Set to -1 if the image width to be acquired is unknown (such as when using a hand-held scanner and dragging left-to-right). In this case the Source must transfer the image in tiles.

ImageLength

Set to -1 if the image length to be acquired is unknown (such as when using a hand-held scanner and dragging top-to-bottom).

If the ImageWidth or ImageHeight is -1, it is implied that the application will receive MSG_XFERREADY before the scan starts.

Since many applications do not handle ImageWidth and/or ImageHeight equal to -1 cases very well, it is recommended that sources use this operation to also trigger the acquisition of the scanned image (e.g., sheet-feed scanners might trigger the start of scan and buffer the data in anticipation of a future IMAGExxxXFER call, returning the correct ImageWidth and ImageHeight for this operation). A sheet-feed scanner may start scan in No UI mode if paper is detected. A handheld scanner will start scanning in UI mode when the user pushes the Start button. "Start scan" means that the normal protocol will be followed but the applications ability to supply buffers may be controlling the start of the scan as well as the speed of the scan. It is really the same events that triggers the scan in both cases.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST     /* No such Source in-session with */
                    /* application                    */
   TWCC_SEQERROR    /* Operation invoked in invalid   */
                    /* state                          */
```

See Also

- DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET
  DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET
  DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET
- Capabilities - ICAP_BITDEPTH, ICAP_COMPRESSION, ICAP_PIXELTYPE, ICAP_PLANARCHUNKY, ICAP_XRESOLUTION, ICAP_YRESOLUTION

DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGELAYOUT, MSG_GET, pImageLayout);
```

pImageLayout = A pointer to a TW_IMAGELAYOUT structure

Valid States

4 through 6

Description

The DAT_IMAGELAYOUT operations control information on the physical layout of the image on the acquisition platform of the Source (e.g. the glass of a flatbed scanner, the size of a photograph, etc.).

The MSG_GET operation describes both the size and placement of the image on the original "page". The coordinates on the original page and the extents of the image are expressed in the unit of measure currently negotiated for ICAP_UNITS (default is inches).

The outline of the image is expressed by a "frame." The Left, Top, Right, and Bottom edges of the frame are stored in pImageLayout->Frame. These values place the frame within the original page. All measurements are relative to the page's "upper-left" corner. Define "upper-left" by how the image would appear on the computer's screen before any rotation or other position transform is applied to the image data. This origin point will be apparent for most Sources (although folks working with satellites or radio telescopes may be at a bit of a loss).

Finally pImageLayout optionally includes information about which frame on the page, which page within a document, and which document the image belongs to. These fields were included mostly for future versions which could merge more than one type of data. A more immediate use might be for an application that needs to keep track of which frame on the page an image came from while acquiring from a Source that can supply more than one image from the same page at the same time. The information in this structure always describes the current image. To set multiple frames for any page simultaneously, reference ICAP_FRAMES.

Application

No special set up or action required, unless the current units of measure are unacceptable. In that case, the application must re-negotiate ICAP_UNITS prior to invoking this operation. Remember to do this in State 4—the only state wherein capabilities can be set or reset.

Beyond supplying possibly interesting position information on the image to be transferred, the application can use this structure to constrain the final size of the image and to relate the image within a series of pages or documents (see the DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET operation).

Source

Fill all fields of pImageLayout. Most Sources will set FrameNumber, PageNumber, and DocumentNumber to 1.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST        /* No such Source in-session    */
                       /* with application             */
   TWCC_SEQERROR       /* Operation invoked in invalid */
                       /* state                        */
```

See Also

- DG_IMAGE / DAT_IMAGELAYOUT / MSG_GETDEFAULT
  DG_IMAGE / DAT_IMAGELAYOUT / MSG_RESET
  DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET
- Capabilities - Many such as ICAP_FRAMES, ICAP_MAXFRAMES, ICAP_UNITS

DG_IMAGE / DAT_IMAGELAYOUT / MSG_GETDEFAULT

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGELAYOUT, MSG_GETDEFAULT,
pImageLayout);
```

pImageLayout = A pointer to a TW_IMAGELAYOUT structure

Valid States

4 through 6

Description

The DAT_IMAGELAYOUT operations control information on the physical layout of the image on the acquisition platform of the Source (e.g. the glass of a flatbed scanner, the size of a photograph, etc.).

This operation returns the default information on the layout of an image. This is the size and position of the image that will be acquired from the Source if the acquisition is started with the Source (and the device it is controlling) in its power-on state (for instance, most flatbed scanners will capture the entire bed).

Application

No special set up or action required.

Source

Fill in all fields of pImageLayout with the device's power-on origin and extents. Most Sources will set FrameNumber, PageNumber, and DocumentNumber to 1.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST        /* No such Source in-session    */
                       /* with application             */
   TWCC_SEQERROR       /* Operation invoked in invalid */
                       /* state                        */
```

See Also

- DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
  DG_IMAGE / DAT_IMAGELAYOUT / MSG_GETDEFAULT
  DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET
- Capabilities - ICAP_FRAMES, ICAP_MAXFRAMES, ICAP_UNITS

DG_IMAGE / DAT_IMAGELAYOUT / MSG_RESET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGELAYOUT, MSG_RESET, pImageLayout);
```

pImageLayout = A pointer to a TW_IMAGELAYOUT structure

Valid States

4 only

Description

The DAT_IMAGELAYOUT operations control information on the physical layout of the image on the acquisition platform of the Source (e.g. the glass of a flatbed scanner, the size of a photograph, etc.).

This operation sets the image layout information for the next transfer to its default settings.

Application

No special set up or action required. Ascertain the current settings of ICAP_ORIENTATION, ICAP_PHYSICALWIDTH, and ICAP_PHYSICALHEIGHT if you don't already know this device's power-on default values.

Source

Reset all the fields of the structure pointed at by pImageLayout to the device's power-on origin and extents. There is an implied resetting of ICAP_ORIENTATION, ICAP_PHYSICALWIDTH, and ICAP_PHYSICALHEIGHT to the device's power-on default values.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADDEST      /* No such Source in-session    */
                     /* with application             */
   TWCC_SEQERROR     /* Operation invoked in invalid */
                     /* state                        */
```

See Also

- DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
  DG_IMAGE / DAT_IMAGELAYOUT / MSG_GETDEFAULT
  DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET
- Capabilities - ICAP_FRAMES, ICAP_MAXFRAMES, ICAP_UNITS

DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGELAYOUT, MSG_SET, pImageLayout);
```

pImageLayout = A pointer to a TW_IMAGELAYOUT structure

Valid States

4 only

Description

The DAT_IMAGELAYOUT operations control information on the physical layout of the image on the acquisition platform of the Source (e.g. the glass of a flatbed scanner, the size of a photograph, etc.).

This operation sets the layout for the next image transfer. This allows the application to specify the physical area to be acquired during the next image transfer (for instance, a frame-based application would pass to the Source the size of the frame the user selected within the application—the helpful Source would present a selection region already sized to match the layout frame size).

If the application and Source have negotiated one or more frames through ICAP_FRAMES, the frame set with this operation will only persist until the transfer following this one. Otherwise, the frame will persist as the current frame for the remainder of the session (unless superseded by negotiation on ICAP_FRAMES or another operation on DAT_IMAGELAYOUT overrides it).

The application writer should note that setting these values is a request. The Source should first try to match the requested values exactly. Failing that, it should approximate the requested values as closely as it can—extents of the approximated frame should at least equal the requested extents unless the device cannot comply. The Source should return TWRC_CHECKSTATUS if the actual values set in pImageLayout->Frame are greater than or equal to the requested values in both extents. If one or both of the requested values exceed the Source's available values, the Source should return TWRC_FAILURE with TWCC_BADVALUE. The application should check for these return codes and perform a MSG_GET to verify that the values set by the Source are acceptable. The application may choose to cancel the transfer if Source could not set the layout information closely enough to the requested values.

Application

Fill in all fields of pImageLayout. Especially important is the Frame field whose values are expressed in ICAP_UNITS. If the application doesn't care about one or more of the other fields, be sure to set them to -1 to prevent confusion. If the application only cares about the extents of the Frame, and not about the origin on the page, set the Frame.Top and Frame.Left to zero. Otherwise, the application can specify the location on the page where the Source should begin acquiring the image, in addition to the extents of the acquired image.

Source

Use the values in pImageLayout as the Source's current image layout information. If you are unable to set the device exactly to the values requested in the Frame field, set them as closely as possible, always snapping to a value that will result in a larger frame, and return TWRC_CHECKSTATUS to the application.

If the application has set Frame.Top and Frame.Left to a non-zero value , set the origin for the image to be acquired accordingly. If possible, the Source should consider reflecting these settings in the user interface when it is raised. For instance, if your Source presents a pre-scan image, consider showing the selection region in the proper location and with the proper size suggested by the settings from this operation.

If the requested values exceed the maximum size the Source can acquire, set the pImageLayout->Frame values used within the Source to the largest extent possible within the axis of the offending value. Return TWRC_FAILURE with TWCC_BADVALUE.

Return Codes

```
TWRC_SUCCESS
TWRC_CHECKSTATUS     /* Source approximated the requested*/
                     /* values                           */
TWRC_FAILURE
   TWCC_BADDEST      /* No such Source in-session       */
                     /* with application                */
   TWCC_BADVALUE     /* Specified Layout values illegal */
                     /* for Source                      */
   TWCC_SEQERROR     /* Operation invoked in invalid    */
                     /* state                           */
```

See Also

- DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
  DG_IMAGE / DAT_IMAGELAYOUT / MSG_GETDEFAULT
  DG_IMAGE / DAT_IMAGELAYOUT / MSG_RESET
- Capabilities - ICAP_FRAMES, ICAP_MAXFRAMES, ICAP_UNITS

DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGEMEMXFER, MSG_GET, pImageMemXfer);
```

pImageMemXfer = A pointer to a TW_IMAGEMEMXFER structure

Valid States

6 and 7 (Transitions to State 7, if successful. Remains in State 7 until MSG_ENDXFER operation.)

Description

This operation is used to initiate the transfer of an image from the Source to the application via the Buffered Memory transfer mechanism.

This operation supports the transfer of successive blocks of image data (in strips or, optionally, tiles) from the Source into one or more main memory transfer buffers. These buffers (for strips) are allocated and owned by the application. For tiled transfers, the source allocates the buffers. The application should repeatedly invoke this operation while TWRC_SUCCESS is returned by the Source.

Application

The application will allocate one or more memory buffers to contain the data being transferred from the Source. The application may allocate enough buffer space to contain the entire image being transferred or, more commonly, use the transfer buffer(s) as a temporary holding area while the complete image is assembled elsewhere (on disk, for instance).

The size of the allocated buffer(s) should be homogeneous (don't change buffer sizes during transfer). The size the application selects should be based on the information returned by the Source from the DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET operation. The application should do its best to allocate transfer buffers of the size "preferred" by the Source. This will enhance the chances for superior transfer performance. The buffer size must be between MinBufSize and MaxBufSize as reported by the Source. Further, the buffers must contain an even number of bytes. Memory buffers must be double-word aligned and should be padded with zeros at the end of each raster line.

If the application sets up buffers that are either too small or too large, the Source will fail the operation returning TWRC_FAILURE/TWCC_BADVALUE.

Once the buffers have been set up, the application should fill pImageMemXfer->Memory.Length with the actual size (in bytes) of each memory buffer (which are, of course, all the same size).

Windows only—The buffers should be allocated in global memory.

Source

Prior to writing the first buffer, check pImageMemXfer->Memory.Length for the size of the buffer(s) the application has allocated. If the size lies outside the maximum or minimum buffer size communicated to the application during the DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET operation, return TWRC_FAILURE/TWCC_BADVALUE and remain in State 6.

If the buffer is of an acceptable size, fill in all fields of pImageMemXfer except pImageMemXfer->Memory. The Source must write the data block into the buffer referenced by pImageMemXfer->Memory.TheMem. Store the actual number of bytes written into the buffer in pImageMemXfer->BytesWritten. Compressed and tiled data effects how the Source fills in these values.

Return TWRC_SUCCESS after successfully writing each buffer. Return TWRC_CANCEL if the Source needs to terminate the transfer before the last buffer is written (as when the user aborts the transfer from the Source's user interface). Return TWRC_XFERDONE to signal that the last buffer has been written. Following completion of the transfer, either after all the data has been written or the transfer has been canceled, remain in State 7 until explicitly transitioned back to State 6 by the application (DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER).

If TWRC_FAILURE occurred on the first buffer, the session remains in State 6. If failing on a subsequent buffer, the session remains in State 7. The strip whose transfer failed is still pending.

Notes on Memory Usage: Following a canceled transfer, the Source should dispose of the image that was being transferred and assure that any temporary variable and local buffer allocations are eliminated. The Source should be wary of allocating large temporary buffers or variables. Doing so may disrupt or even disable the transfer process. The application should be aware of the possible needs of the Source to allocate such space, however, and consider allocating all large blocks of RAM needed to support the transfer prior to invoking this operation. This may be especially important for devices that create image transfers of indeterminate size—such as hand-held scanners.

Return Codes

```
TWRC_SUCCESS                /* Source done transferring       */
                            /* the specified block            */
TWRC_XFERDONE               /* Source done transferring       */
                            /* the specified image            */
TWRC_CANCEL                 /* User aborted the transfer from */
                            /* the Source                     */
TWRC_FAILURE
   TWCC_BADDEST             /* No such Source in-session */
                            /*with application           */
   TWCC_BADVALUE            /* Size of buffer did not    */
                            /* match TW_SETUPMEMXFER     */
   TWCC_OPERATIONERROR      /* Failure in the Source--   */
                            /* transfer invalid          */
   TWCC_SEQERROR            /* Operation invoked in      */
                            /* invalid state             */
```

See Also

- DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET
  DG_IMAGE / DAT_IMAGEINFO / MSG_GET
  DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET

- Capabilities - ICAP_COMPRESSION, ICAP_TILES, ICAP_XFERMECH

DG_IMAGE / DAT_IMAGENATIVEXFER / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_IMAGENATIVEXFER, MSG_GET, pHandle);
```

pHandle = A pointer to a variable of type TW_UINT32

On Windows - This 32 bit integer is a handle variable to a DIB (Device Independent Bitmap) located in memory.

On Macintosh - This 32-bit integer is a handle to a Picture (a PicHandle). It is a QuickDraw picture located in memory.

Valid States

6 only (Transitions to State 7, if successful. Remains in State 7 until MSG_ENDXFER operation).

Description

Causes the transfer of an image's data from the Source to the application, via the Native transfer mechanism, to begin. The resulting data is stored in main memory in a single block. The data is stored in Picture (PICT) format on the Macintosh and as a device-independent bitmap (DIB) under MS Windows. The size of the image that can be transferred is limited to the size of the memory block that can be allocated by the Source.

Note: This is the default transfer mechanism. All Source's support this mechanism. The Source will use this mechanism unless the application explicitly negotiates a different transfer mechanism with ICAP_XFERMECH.

Application

The application need only invoke this operation once per image. The Source allocates the largest block available and transfers the image into it. If the image is too large to fit, the Source may resize the image. Read the DIB header or check the picFrame in the Picture to determine if this happened. The application is responsible for deallocating the memory block holding the Native-format image.

MS Windows only—Set pHandle pointing to a handle to a device-independent bit map (DIB) in memory. The Source will allocate the image buffer and return the handle to the address specified..

Macintosh only—Set pHandle pointing to a handle to a Picture in memory. The Source will allocate the image buffer at the memory location referenced by the handle.

Note: This odd combination of pointer and handle to reference the image data block was used to assure that the allocated memory object would be relocatable under MS Windows, Macintosh, and UNIX. A handle was required for this task on both the Macintosh and under MS Windows; though pointers are inherently relocatable under UNIX. Rather than disturb the entry points convention that the data object is always referenced by a pointer, it was decided to have that pointer reference the relocatable handle. A handle in UNIX is typecast to a pointer.

Source

Allocate a single block of memory to hold the image data and write the image data into it using the appropriate format for the operating environment. The source must assure that the allocated block will be accessible to the application. Place the handle of the allocated block in the TW_UINT32 pointed to by pHandle.

MS Windows: Format the data block as a DIB. Use GlobalAlloc or equivalent under windows. Under 16 bit MS Windows, place the handle in the low word of the TW_UINT32. The following assignment will work in either Win16 or Win32:

(HGLOBAL FAR *) pHandle = hDIB;

See the Windows SDK documentation under Structures: BIMAPINFO, BITMAPINFOHEADER, RGBQUAD. See also "DIBs and their use" by Ron Gery, in the Microsoft Development Library (MSDN CD).

Notes:

- Do not use BITMAPCOREINFO or BIMAPCOREHEADER as these are for OS/2 compatibility only.
- Always follow the BITMAPINFOHEADER with the color table and only save 1, 4, or 8 bit DIBs
- Color table entries are RGBQUADs, which are stored in memory as BGR not RGB.
- For 24 bit color DIBs, the "pixels" are also stored in BGR order, not RGB.
- DIBs are stored 'upside-down' - the first pixel in the DIB is the lower-left corner of the image, and the last pixel is the upper-right corner.
- DIBs can be larger than 64K, but be careful, a 24 bit pixel can straddle a 64K boundary!
- Pixels in 1, 4, and 8 bit DIBs are "always" color table indices, you must index through the color table to determine the color value of a pixel.

Macintosh: Format the data block as a PICT, preferably using standard system calls.

MS Windows and Macintosh: If the allocation fails, it is recommended that you allow the user the option to re-size the image to fit within available memory or to cancel the transfer (assuming that the Source user interface is displayed). If the user chooses to cancel the transfer, return TWRC_CANCEL. If the user wants to re-size the image, the Source might choose to blindly crop the image, clip a selection region to the maximum supported size for the current memory configuration, or allow the user to re-acquire the image altogether. The user will usually feel more in control if you provide one or both of the last two options, but the first may make the most sense for your Source.

If the allocation fails and the image cannot be clipped, return TWRC_FAILURE and remain in State 6. Set the pHandle to NULL. The image whose transfer failed is still pending transfer. Do not decrement TW_PENDINGXFERS.Count.

Return Codes

```
TWRC_XFERDONE            /* Source done transferring the */
                         /* specified block              */
TWRC_CANCEL              /* User aborted the transfer    */
                         /* within the Source            */
TWRC_FAILURE
   TWCC_BADDEST          /* No such Source in session */
                         /* with application          */
   TWCC_LOWMEMORY        /* Not enough memory for     */
                         /* image--cannot crop to fit */
   TWCC_OPERATIONERROR   /* Failure in the Source--   */
                         /* transfer invalid          */
   TWCC_SEQERROR         /* Operation invoked in      */
                         /* invalid state             */
```

See Also

- DG_IMAGE / DAT_IMAGEINFO / MSG_GET
  DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
- Capability - ICAP_XFERMECH

DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_JPEGCOMPRESSION, MSG_GET, pCompData);
```

pCompData = A pointer to a TW_JPEGCOMPRESSION structure

Valid States

4 through 6

Description

Causes the Source to return the parameters that will be used during the compression of data using the JPEG algorithms.

All the information that is reported by the MSG_GET operation will be available in the header portion of the JPEG data. Transferring JPEG-compressed data through memory buffers is slightly different than other types of buffered transfers. The difference is that the JPEG-compressed image data will be prefaced by a block of uncompressed information—the JPEG header. This header information contains all the information that is returned from the MSG_GET operation. The compressed image information follows the header. The Source should return the header information in the first transfer. The compressed image data will then follow in the second through the final buffer. If the application is allocating the buffers, it should assure that the buffer size for transfer of the header is large enough to contain the complete header.

Application

The application allocates the TW_JPEGCOMPRESSION structure.

Source

Fill pCompData with the parameters that will be applied to the next JPEG-compression operation. The Source must allocate memory for the contents of the pointer fields pointed to within the structure (i.e. QuantTable, HuffmanDC, and HuffmanAC).

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL       /* Source does not support JPEG */
                          /* data compression             */
   TWCC_SEQERROR          /* Operation invoked in invalid */
                          /* state                        */
```

See Also

- DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GETDEFAULT
  DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_RESET
  DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_SET
- Capability - ICAP_COMPRESSION

DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GETDEFAULT

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_JPEGCOMPRESSION, MSG_GETDEFAULT,
pCompData);
```

pCompData = A pointer to a TW_JPEGCOMPRESSION structure

Valid States

4 through 6

Description

Causes the Source to return the power-on default values applied to JPEG-compressed data transfers.

Application

The application allocates the TW_JPEGCOMPRESSION structure.

Source

Fill in pCompData with the power-on default values. The Source must allocate memory for the contents of the pointer fields pointed to within the structure (i.e. QuantTable, HuffmanDC and HuffmanAC). The Source should maintain meaningful default values.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL        /* Source does not support JPEG */
                           /* data compression             */
   TWCC_SEQERROR           /* Operation invoked in invalid */
                           /* state                        */
```

See Also

- DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GET
  DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_RESET
  DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_SET
- Capability - ICAP_COMPRESSION

DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_RESET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_JPEGCOMPRESSION, MSG_RESET,
pCompData);
```

pCompData = A pointer to a TW_JPEGCOMPRESSION structure

Valid States

4 only

Description

Return the Source to using its power-on default values for JPEG-compressed transfers.

Application

No special action. May want to perform a MSG_GETDEFAULT if you're curious what the new values might be.

Source

Use your power-on default values for all future JPEG-compressed transfers. The Source should maintain meaningful default values for all parameters.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL       /* Source does not support JPEG */
                          /* data compression             */
   TWCC_SEQERROR          /* Operation invoked in invalid */
                          /* state                        */
```

See Also

- DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GET
  DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GETDEFAULT
  DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_SET
- Capability - ICAP_COMPRESSION

DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_SET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_JPEGCOMPRESSION, MSG_SET, pCompData);
```

pCompData = A pointer to a TW_JPEGCOMPRESSION structure

Valid States

4 only

Description

Allows the application to configure the compression parameters to be used on all future JPEG-compressed transfers during the current session. The application should have already established that the requested values are supported by the Source.

Application

Fill pCompData. Write TWON_DONTCARE16 into the numeric fields that don't matter to the application. Write NULL into the table fields that should use the default tables as defined by the JPEG specification.

Source

Adopt the requested values for use with all future JPEG-compressed transfers. If a value does not exactly match an available value, match the value as closely as possible and return TWRC_CHECKSTATUS. If the value is beyond the range of available values, clip to the nearest value and return TWRC_FAILURE/TWCC_BADVALUE.

Return Codes

```
TWRC_SUCCESS
TWRC_CHECKSTATUS
TWRC_FAILURE
   TWCC_BADPROTOCOL         /* Source does not support JPEG */
                            /* data compression             */
   TWCC_BADVALUE            /* illegal value specified      */
   TWCC_SEQERROR            /* Operation invoked in invalid */
                            /* state                        */
```

See Also

- DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GET
  DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GETDEFAULT
  DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_RESET
- Capability - ICAP_COMPRESSION

DG_IMAGE / DAT_PALETTE8 / MSG_GET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_PALETTE8, MSG_GET, pPalette);
```

pPalette = A pointer to a TW_PALETTE8 structure

Valid States

4 through 6

Description

This operation causes the Source to report its current palette information. The application should assure that the Source can provide palette information by invoking a MSG_GET operation on ICAP_PIXELTYPE and checking for TWPT_PALETTE. If this pixel type has not been established as the type to be used for future acquisitions, the Source should respond with its default palette.

To assure that the palette information is wholly accurate, the application should invoke this operation immediately after completion of the image transfer. The Source may perform palette optimization during acquisition of the data and the palette it reports before the transfer will differ from the one available afterwards.

(In general, the DAT_PALETTE8 operations are specialized to deal with 8-bit data, whether grayscale or color (8-bit or 24-bit). Most current devices provide data with this bit depth. These operations allow the application to inquire a Source's support for palette color data and set up a palette color transfer. See Chapter 8 for the definitions and data structures used to describe palette color data within TWAIN.)

Application

The application should allocate the pPalette structure for the Source.

Source

Fill pPalette with the current palette. If no palette has been specified or calculated, use the Source's default palette (which may coincidentally be the current or default system palette).

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL          /* Source does not support      */
                             /* palette color transfers      */
   TWCC_SEQERROR             /* Operation invoked in invalid */
                             /* state                        */
```

See Also

- DG_IMAGE / DAT_PALETTE8 / MSG_GETDEFAULT
  DG_IMAGE / DAT_PALETTE8 / MSG_RESET
  DG_IMAGE / DAT_PALETTE8 / MSG_SET
- Capability - ICAP_PIXELTYPE

DG_IMAGE / DAT_PALETTE8 / MSG_GETDEFAULT

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_PALETTE8, MSG_GETDEFAULT, pPalette);
```

pPalette = A pointer to a TW_PALETTE8 structure

Valid States

4 through 6

Description

This operation causes the Source to report its power-on default palette.

Application

The application should allocate the pPalette structure for the Source.

Source

Fill pPalette with the default palette.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL        /* Source does not support      */
                           /* palette color transfers      */
   TWCC_SEQERROR           /* Operation invoked in invalid */
                           /* state                        */
```

See Also

- DG_IMAGE / DAT_PALETTE8 / MSG_GET
  DG_IMAGE / DAT_PALETTE8 / MSG_RESET
  DG_IMAGE / DAT_PALETTE8 / MSG_SET
- Capability - ICAP_PIXELTYPE

DG_IMAGE / DAT_PALETTE8 / MSG_RESET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_PALETTE8, MSG_RESET, pPalette);
```

pPalette = A pointer to a TW_PALETTE8 structure

Valid States

4 only

Description

This operation causes the Source to dispose of any current palette it has and to use its default palette for the next palette transfer. A Source that always performs palette optimization may not use the default palette for the next transfer, but should dispose of its current palette and adopt the default palette for the moment, anyway. The application can check the actual palette information by invoking a MSG_GET operation immediately following the image transfer.

Application

The application should allocate the pPalette structure for the Source.

Source

Fill pPalette with the default palette and use the default palette for the next palette transfer.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL          /* Source does not support      */
                             /* palette color transfers      */
   TWCC_SEQERROR             /* Operation invoked in invalid */
                             /* state                        */
```

See Also

- DG_IMAGE / DAT_PALETTE8 / MSG_GET
  DG_IMAGE / DAT_PALETTE8 / MSG_GETDEFAULT
  DG_IMAGE / DAT_PALETTE8 / MSG_SET
- Capability - ICAP_PIXELTYPE

DG_IMAGE / DAT_PALETTE8 / MSG_SET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_PALETTE8, MSG_SET, pPalette);
```

pPalette = A pointer to a TW_PALETTE8 structure

Valid States

4 only

Description

This operation requests that the Source adopt the specified palette for use with all subsequent palette transfers. The application should be careful to supply a palette that matches the bit depth of the Source. The Source is not required to adopt this palette. The application should be careful to check the return value from this operation.

Application

Fill pPalette with the desired palette. If writing grayscale information, write the same data into the Channel1, Channel2, and Channel3 fields of the Colors array. If NumColors != 256, fill the unused array elements with minimum ("black") values.

Source

The Source should not return TWRC_SUCCESS unless it will actually use the requested palette. The Source should not modify the palette in any way until the transfer is complete. The palette should be used for all remaining palette transfers for the duration of the session.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL          /* Source does not support      */
                             /* palette color transfers      */
   TWCC_SEQERROR             /* Operation invoked in invalid */
                             /* state                        */
```

See Also

- DG_IMAGE / DAT_PALETTE8 / MSG_GET
  DG_IMAGE / DAT_PALETTE8 / MSG_GETDEFAULT
  DG_IMAGE / DAT_PALETTE8 / MSG_RESET
- Capability - ICAP_PIXELTYPE

DG_IMAGE / DAT_RGBRESPONSE / MSG_RESET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_RGBRESPONSE, MSG_RESET, pResponse);
```

pResponse = A pointer to a TW_RGBRESPONSE structure

Valid States

4 only

Description

Causes the Source to use its "identity" response curves for future RGB transfers. The identity curve causes no change in the values of the captured data when it is applied. (Note that resetting the curves for RGB data **does not** reset any MSG_SET curves for other pixel types).

Note: The DAT_RGBRESPONSE operations allow the application to specify the transfer curves that the Source should apply to the RGB data it acquires. The Source should not support these operations unless it can provide data of pixel type TWPT_RGB. The Source need not maintain actual "identity response curves" for use with the MSG_RESET operation—once reset, the Source should transfer the RGB data as acquired from the Source. The application should be sure that the Source supports these operations before invoking them. The operations should only be invoked when the active pixel type is RGB (TWPT_RGB). See Chapter 8 for information about the definitions and data structures used to describe the RGB response curve within TWAIN.

Application

No special action.

Source

Apply the identity response curve to all future RGB transfers. This means that the Source will transfer the RGB data exactly as acquired from the device.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL          /* Source does not support RGB  */
                             /* response curves              */
   TWCC_BADVALUE             /* Current pixel type is not    */
                             /* TWPT_RGB                     */
   TWCC_SEQERROR             /* Operation invoked in invalid */
                             /* state                        */
```

See Also

- DG_IMAGE / DAT_RGBRESPONSE / MSG_SET
- Capability - ICAP_PIXELTYPE

DG_IMAGE / DAT_RGBRESPONSE / MSG_SET

Call

```
DSM_Entry(pOrigin, pDest, DG_IMAGE, DAT_RGBRESPONSE, MSG_SET, pResponse);
```

pResponse = A pointer to a TW_RGBRESPONSE structure

Valid States

4 only

Description

Causes the Source to transform any RGB data according to the response curves specified by the application.

Application

Fill all three elements of the response curve with the response curve data you want the Source to apply to future RGB transfers. The application should consider writing the same values into each element of the same index to minimize color shift problems.

The Source may not support this operation. The application should ensure that the current pixel type is TWPT_RGB and examine the return code from this operation.

Source

Apply the specified response curves to all future RGB transfers.

Return Codes

```
TWRC_SUCCESS
TWRC_FAILURE
   TWCC_BADPROTOCOL          /* Source does not support color */
                             /* response curves               */
   TWCC_BADVALUE             /* Current pixel type is not RGB */
   TWCC_SEQERROR             /* Operation invoked in invalid  */
                             /* state                         */
```

See Also

- DG_IMAGE / DAT_RGBRESPONSE / MSG_RESET
- Capability - ICAP_PIXELTYPE

8

Data Types and Data Structures

Chapter Contents

TWAIN defines a large number of data types and structures. These are all defined in the TWAIN.H file that is shipped as part of this toolkit. The file is written in C so you will need to modify the syntax if you develop your application or Source in some other language.

Naming Conventions

Data Structures, Variables, Pointers and Handles

Data structures referenced by pData parameter in DSM_Entry calls

Are prefixed by TW_ and followed by a descriptive name, in upper case. The name typically matches the call's DAT parameter.

Example: TW_USERINTERFACE

Fields in data structures (not containing pointers or handles)

Typically, begin with a capital letter followed by mixed upper and lower case letters.

Example: The MinBufSize, MaxBufSize, and Preferred fields in which are in the TW_SETUPMEMXFER structure.

Fields in data structures that contain pointers or handles

Name starts with lower case "p" or "h" for pointer or handle followed by a typical field name with initial capital then mixed case characters.

Example: pData, hContainer

Constants and Types

General-use constants

Are prefixed by TWON_ followed by the description of the constant's meaning.

Example: TWON_ICONID, TWON_ARRAY

Specific-use constants

Are prefixed with TWxx_ where xx are two letters identifying the group to which the constant belongs.

Example: TWTY_INT16, TWTY_STR32 are constants of the group "TW Types"

Common data types

Rather than use the int, char, long, etc. types with their variations between compilers, TWAIN defines a group of types that are used to cast each data item used by the protocol. Types are prefixed and named exactly the same as TWAIN data structures, TW_ followed by a descriptive name, all in upper case characters.

Example: TW_UINT32, TW_HANDLE

Custom Constants

Applications and Sources may define their own private (custom) constant identifiers for any existing constant group by assigning the constant a value greater than or equal to 256. They may also define any new desired custom constant group. The consuming entity should check the originating entity's TW_IDENTITY.ProductName when encountering a constant value greater than or equal to 256 to see whether it can be recognized as a custom constant. Sources and applications should not assume that all entities will have such error checking built in, however.

The following are operation identifiers:

Data Groups	Prefixed with DG_
Data Argument Types	Prefixed with DAT_
Messages	Prefixed with MSG_
Return codes	Prefixed with TWRC_
Condition codes	Prefixed with TWCC_
General capabilities	Prefixed with CAP_
Image-specific capabilities	Prefixed with ICAP_

As a general note, whenever the application or the Source allocates a TWAIN data structure, it should fill all the fields it is instructed to fill and write the default value (if one is specified) into any field it is not filling. If no default is specified, fill the field with the appropriate TWON_DONTCARExx constant where xx describes the size of the field in bits (bytes, in the case of strings). The TWON_ constants are described at the end of this chapter and defined in the TWAIN.H file.

Some fields return a value of -1 when the data to be returned is ambiguous or unknown. Applications and Sources must look for these special cases, especially when allocating memory. Examples of Fields with -1 values are found in TW_PENDINGXFERS (Count), TW_SETUPMEMXFER (MaxBufSize) and TW_IMAGEINFO (ImageWidth and ImageLength).

The remainder of this chapter lists the defined data types and data structures. Most of the constants are also listed. However, refer to the TWAIN.H file for more explanation about each constant and to see the lengthy list of country constants which are not duplicated here.

Platform Dependent Definitions and Typedefs

On Windows

```
typedef HANDLE          TW_HANDLE;
typedef LPVOID          TW_MEMREF;
```

On Macintosh

```
#define PASCAL          pascal
#define FAR
typedef Handle          TW_HANDLE;
typedef char            *TW_MEMREF;
```

On UNIX

```
#define PASCAL          pascal
typedef unsigned char   *TW_HANDLE;
typedef unsigned char   *TW_MEMREF;
```

Definitions of Common Types

String types

```
typedef char        TW_STR32[34],       FAR *pTW_STR32;
typedef char        TW_STR64[66],       FAR *pTW_STR64;
typedef char        TW_STR128[130],     FAR *pTW_STR128;
typedef char        TW_STR255[256],     FAR *pTW_STR255;
```

On Windows: These include room for the strings and a NULL character.

On Macintosh: These include room for a length byte followed by the string.

Note: he TW_STR255 must hold less than 256 characters so the length fits in the first byte on Macintosh.

Numeric types

```
typedef char            TW_INT8     FAR *pTW_INT8;
typedef short           TW_INT16    FAR *pTW_INT16;
typedef long            TW_INT32    FAR *pTW_INT32;
typedef unsigned char   TW_UINT8    FAR *pTW_UINT8;
typedef unsigned short  TW_UINT16   FAR *pTW_UINT16;
typedef unsigned long   TW_UINT32   FAR *pTW_UINT32;
typedef unsigned short  TW_BOOL     FAR *pTW_BOOL;
```

Fixed point structure type

```
typedef struct {
        TW_INT16        Whole;
        TW_UINT16       Frac;
} TW_FIX32,     FAR *pTW_FIX32;
```

Note: In cases where the data type is smaller than TW_UINT32, the data should reside in the lower word.

Data Structure Definitions

This section provides descriptions of the data structure definitions.

TW_ARRAY

```
typedef struct {
   TW_UINT16       ItemType;
```

```
   TW_UINT32       NumItems;
   TW_UINT8        ItemList[1];
} TW_ARRAY, FAR * pTW_ARRAY;
```

Used by

TW_CAPABILITY structure (when ConType field specifies TWON_ARRAY)

Description

This structure stores a group of associated individual values which, when taken as a whole, describes a single "value" for a capability. The values need have no relationship to one another aside from being used to describe the same "value" of the capability. Such an array of values is useful to describe the CAP_SUPPORTEDCAPS list. This structure is used as a member of TW_CAPABILITY structures. Since this structure does not, therefore, exist "stand-alone" it is identified by a TWON_xxxx constant rather than a DAT_xxxx. This structure is related in function and purpose to TW_ENUMERATION, TW_ONEVALUE, and TW_RANGE.

Field Descriptions

ItemType	The type of items in the array. The type is indicated by the constant held in this field. The constant is of the kind TWTY_xxxx. All items in the array have the same size.
NumItems	How many items are in the array.
ItemList[1]	This is the array. One value resides within each element of the array. Space for the array is not allocated inside this structure. The ItemList value is simply a placeholder for the start of the actual array, which must be allocated when the container is allocated . Remember to typecast the allocated array, as well as references to the elements of the array, to the type indicated by the constant in ItemType.

TW_CAPABILITY

```
typedef struct {
   TW_UINT16       Cap;
   TW_UINT16       ConType;
   TW_HANDLE       hContainer;
} TW_CAPABILITY, FAR * pTW_CAPABILITY;
```

Used by

DG_CONTROL / DAT_CAPABILITY / MSG_GET
DG_CONTROL / DAT_CAPABILITY / MSG_GETCURRENT
DG_CONTROL / DAT_CAPABILITY / MSG_GETDEFAULT
DG_CONTROL / DAT_CAPABILITY / MSG_RESET
DG_CONTROL / DAT_CAPABILITY / MSG_SET

Description

Used by an application either to get information about, or control the setting of a capability. The first field identifies the capability being negotiated (e.g., ICAP_BRIGHTNESS). The second specifies the format of the container (e.g., TWON_ONEVALUE). The third is a handle (HGLOBAL under MS Windows) to the container itself.

The application always sets the Cap field. On MSG_SET, the application also sets the ConType and hContainer fields. On MSG_RESET, MSG_GET, MSG_GETCURRENT, and MSG_GETDEFAULT, the source fills in the ConType and hContainer fields.

It is always the application's responsibility to free the container when it is no longer needed. On a MSG_GET, MSG_GETCURRENT, or MSG_GETDEFAULT, the source allocates the container but ownership passes to the application. On a MSG_SET, the application provides the container either by allocating it or by re-using a container created earlier.

On a MSG_SET, the Source must not modify the container and it must copy any data that it wishes to retain.

Field Descriptions

Cap	The numeric designator of the capability (of the form CAP_xxxx or ICAP_xxx). e.g. ICAP_BRIGHTNESS. A list of these can be found in Chapter 9 and in the TWAIN.H file.
ConType	The type of the container referenced by hContainer. The container structure will be one of four types: TWON_ARRAY, TWON_ENUMERATION, TWON_ONEVALUE, or TWON_RANGE. One of these values, which types the container, should be entered into this field by whichever TWAIN entity fills in the container. When the application wants to set (MSG_SET) the Source's capability, the application must fill in this field. When the application wants to get (MSG_GET) capability information from the Source, the Source must fill in this field.
hContainer	References the container structure where detailed information about the capability is stored. When the application wants to set (MSG_SET) the Source's capability, the application must provide the hContainer. When the application wants to get (MSG_GET) the Source's capability information, the Source must allocate the space for the container. In either case, the application must release this space.

TW_CIECOLOR

```
typedef struct {
   TW_UINT16           ColorSpace
   TW_INT16            LowEndian;
   TW_INT16            DeviceDependent;
   TW_INT32            VersionNumber;
   TW_TRANSFORMSTAGE   StageABC;
   TW_TRANSFORMSTAGE   StageLMN;
   TW_CIEPOINT         WhitePoint;
   TW_CIEPOINT         BlackPoint;
   TW_CIEPOINT         WhitePaper;
   TW_CIEPOINT         BlackInk;
   TW_FIX32            Samples[1];
} TW_CIECOLOR, FAR *  pTW_CIECOLOR;
```

Used by

DG_IMAGE / DAT_CIECOLOR / MSG_GET

Description

Defines the mapping from an RGB color space device into CIE 1931 (XYZ) color space. For more in-depth information, please reference Appendix A and the PostScript Language Reference Manual, Second Edition, pp. 176-93. Note that the field names do not follow the conventions used elsewhere within TWAIN. This breach allows the identifiers shown here to exactly match those described in Appendix A, which was not written specifically for this Toolkit. Please also note that ColorSpace has been redefined from its form in Appendix A to use TWPT_xxxx constants defined in the TWAIN.H file.

This structure closely parallels the TCIEBasedColorSpace structure definition in Appendix A. Note that the field names are slightly different and that two new fields have been added (WhitePaper and BlackInk) to describe the reflective characteristics of the page from which the image was acquired.

If the Source can provide TWPT_CIEXYZ, it must support all operations on this structure.

Field Descriptions

ColorSpace	Defines the original color space that was transformed into CIE XYZ. Use a constant of type TWPT_xxxx. This value is not set-able by the application. Application should write TWON_DONTCARE16 into this on a MSG_SET.
LowEndian	Used to indicate which data byte is taken first. If zero, then high byte is first. If non-zero, then low byte is first.
DeviceDependent	If non-zero then color data is device-dependent and only ColorSpace is valid in this structure.
VersionNumber	Version of the color space descriptor specification used to define the transform data. The current version is zero.
StageABC	Describes parametrics for the first stage transformation of the Postscript Level 2 CIE color space transform process. Refer to Figure 1, Appendix A.
StageLMN	Describes parametrics for the first stage transformation of the Postscript Level 2 CIE color space transform process. Refer to Figure 1, Appendix A.
WhitePoint	Values that specify the CIE 1931 (XYZ space) tri-stimulus value of the diffused white point.
BlackPoint	Values that specify the CIE 1931 (XYZ space) tri-stimulus value of the diffused black point.
WhitePaper	Values that specify the CIE 1931 (XYZ space) tri-stimulus value of ink-less “paper” from which the image was acquired.
BlackInk	Values that specify the CIE 1931 (XYZ space) tri-stimulus value of solid black ink on the “paper” from which the image was acquired.
Samples[1]	Optional table look-up values used by the decode function. Samples are ordered sequentially and end-to-end as A, B, C, L, M, and N.

TW_CIEPOINT

```
typedef struct {
   TW_FIX32       X;
   TW_FIX32       Y;
   TW_FIX32       Z;
} TW_CIEPOINT, FAR * pTW_CIEPOINT;
```

Used by

Embedded in the TW_CIECOLOR structure

Description

Defines a CIE XYZ space tri-stimulus value. This structure parallels the TCIEPoint structure definition in Appendix A.

Field Descriptions

X First tri-stimulus value of the CIE space representation.

Y Second tri-stimulus value of the CIE space representation.

Z Third tri-stimulus value of the CIE space representation.

TW_DECODEFUNCTION

```
typedef struct {
   TW_FIX32       StartIn;
   TW_FIX32       BreakIn;
   TW_FIX32       EndIn;
   TW_FIX32       StartOut;
   TW_FIX32       BreakOut;
   TW_FIX32       EndOut;
   TW_FIX32       Gamma;
   TW_FIX32       SampleCount;
} TW_DECODEFUNCTION, FAR * pTW_DECODEFUNCTION;
```

Used by

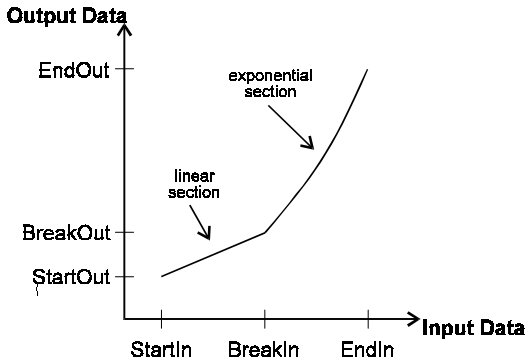
Embedded in the TW_TRANSFORMSTAGE structure that is embedded in the TW_CIECOLOR structure

Description

Defines the parameters used for channel-specific transformation. The transform can be described either as an extended form of the gamma function or as a table look-up with linear interpolation. This structure parallels the TDecodeFunction structure definition in Appendix A.

Field Descriptions

StartIn	Starting input value of the extended gamma function. Defines the minimum input value of channel data.
BreakIn	Ending input value of the extended gamma function. Defines the maximum input value of channel data.
EndIn	The input value at which the transform switches from linear transformation/interpolation to gamma transformation.
StartOut	Starting output value of the extended gamma function. Defines the minimum output value of channel data.
BreakOut	Ending output value of the extended gamma function. Defines the maximum output value of channel data.
EndOut	The output value at which the transform switches from linear transformation/interpolation to gamma transformation.
Gamma	Constant value. The exponential used in the gamma function.
SampleCount	The number of samples in the look-up table. Includes the values of StartIn and EndIn. Zero-based index (actually, number of samples - 1). If zero, use extended gamma, otherwise use table look-up.



Extended Gamma Parameters

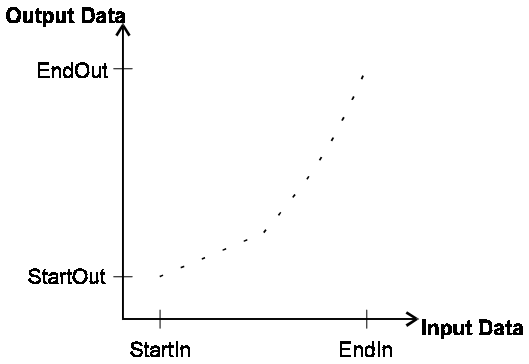


Table Look-up Parameters

TW_ELEMENT8

```
typedef struct {
   TW_UINT8       Index;
   TW_UINT8      Channel1;
   TW_UINT8       Channel2;
   TW_UINT8       Channel3;
 } TW_ELEMENT8, FAR * pTW_ELEMENT8;
```

Used by

Embedded in the TW_GRAYRESPONSE, TW_PALETTE8 and TW_RGBRESPONSE structures

Description

This structure holds the tri-stimulus color palette information for TW_PALETTE8 structures. The order of the channels shall match their alphabetic representation. That is, for RGB data, R shall be channel 1. For CMY data, C shall be channel 1. This allows the application and Source to maintain consistency. Grayscale data will have the same values entered in all three channels.

Field Descriptions

Index	Value used to index into the color table. Especially useful on the Macintosh.
Channel1	First tri-stimulus value (e.g. Red).
Channel2	Second tri-stimulus value (e.g. Green).
Channel3	Third tri-stimulus value (e.g. Blue).

TW_ENUMERATION

```
typedef struct {
  TW_UINT16      ItemType;
  TW_UINT32      NumItems;
  TW_UINT32       CurrentIndex;
  TW_UINT32      DefaultIndex;
  TW_UINT8       ItemList[1];
} TW_ENUMERATION, FAR * pTW_ENUMERATION;
```

Used by

TW_CAPABILITY structure (when ConType field specifies TWON_ENUMERATION)

Description

Stores a group of individual values describing a capability. The values are ordered from lowest to highest values, but the step size between each value is probably not uniform. Such a list would be useful to describe the discreet resolutions of a capture device supporting, say, 75, 150, 300, 400, and 800 dots per inch.

This structure is related in function and purpose to TW_ARRAY, TW_ONEVALUE, and TW_RANGE.

Field Descriptions

ItemType	The type of items in the enumerated list. The type is indicated by the constant held in this field. The constant is of the kind TWTY_xxxx. All items in the array have the same size.
NumItems	How many items are in the enumeration.
CurrentIndex	The item number, or index (zero-based) into ItemList[], of the "current" value for the capability.
DefaultIndex	The item number, or index (zero-based) into ItemList[], of the "power-on" value for the capability.
ItemList[1]	The enumerated list: one value resides within each array element. Space for the list is not allocated inside this structure. The ItemList value is simply a placeholder for the start of the actual array, which must be allocated when the container is allocated. Remember to typecast the allocation to ItemType, as well as references to the elements of the array.

TW_EVENT

```
typedef struct {
   TW_MEMREF      pEvent;
   TW_UINT16      TWMessage;
TW_EVENT, FAR * pTW_EVENT;
```

Used by

DG_CONTROL / DAT_EVENT / MSG_PROCESSEVENT

Description

Used to pass application events/messages from the application to the Source. The Source is responsible for examining the event/message, deciding if it belongs to the Source, and returning an appropriate return code to indicate whether or not the Source owns the event/message. This process is covered in more detail in the Event Loop section of Chapter 3.

Field Descriptions

pEvent	A pointer to the event/message to be examined by the Source. Under MS Windows, pEvent is a pMSG (pointer to an MS Windows MSG struct). That is, the message the application received from GetMessage(). On the Macintosh, pEvent is a pointer to an EventRecord.
TWMessage	Any message (MSG_xxxx) the Source needs to send to the application in response to processing the event/message. The messages currently defined for this purpose are MSG_NULL, MSG_XFERREADY and MSG_CLOSEDSREQ.

TW_FIX32

```
typedef struct {
   TW_INT16       Whole;
   TW_UINT16      Frac;
} TW_FIX32, FAR * pTW_FIX32;
```

Used by

Embedded in the TW_CIECOLOR, TW_CIEPOINT, TW_DECODEFUNCTION, TW_FRAME, TW_IMAGEINFO, and TW_TRANSFORMSTAGE structures.

Used in TW_ARRAY, TW_ENUMERATION, TW_ONEVALUE, and TW_RANGE structures when ItemType is TWTY_FIX32.

Description

Stores a Fixed point number in two parts, a whole and a fractional part. The Whole part carries the sign for the number. The Fractional part is unsigned.

Field Descriptions

Whole	The Whole part of the floating point number. This number is signed.
Frac	The Fractional part of the floating point number. This number is unsigned.

The following functions convert TW_FIX32 to float and float to TW_FIX32:

```
/*************************************************************
* FloatToFix32
* Convert a floating point value into a FIX32.
*************************************************************/
TW_FIX32 FloatToFix32 (float floater)
{
   TW_FIX32 Fix32_value;
   TW_INT32 value = (TW_INT32) (floater * 65536.0 + 0.5);
   Fix32_value.Whole = value >> 16;
   Fix32_value.Frac = value & 0x0000ffffL;
   return (Fix32_value);
}
/*************************************************************
* Fix32ToFloat
* Convert a FIX32 value into a floating point value.
*************************************************************/
float FIX32ToFloat (TW_FIX32 fix32)
{
   float     floater;
   floater = (float) fix32.Whole + (float) fix32.Frac / 65536.0;

   return floater;
}
```

TW_FRAME

```
typedef struct {
   TW_FIX32       Left;
   TW_FIX32       Top;
   TW_FIX32       Right;
   TW_FIX32       Bottom;
} TW_FRAME, FAR * pTW_FRAME;
```

Used by

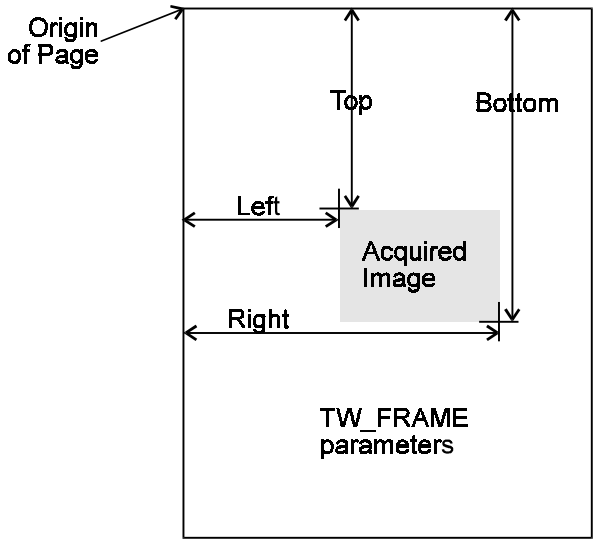
Embedded in the TW_IMAGELAYOUT structure

Description

Defines a frame rectangle in ICAP_UNITS coordinates.

Field Descriptions

Left	Value of the left-most edge of the rectangle (in ICAP_UNITS).
Top	Value of the top-most edge of the rectangle (in ICAP_UNITS).
Right	Value of the right-most edge of the rectangle (in ICAP_UNITS).
Bottom	Value of the bottom-most edge of the rectangle (in ICAP_UNITS).



Frame Parameters

TW_GRAYRESPONSE

```
typedef struct {
   TW_ELEMENT8       Response[1];
} TW_GRAYRESPONSE, FAR * pTW_GRAYRESPONSE;
```

Used by

DG_IMAGE / DAT_GRAYRESPONSE / MSG_RESET
DG_IMAGE / DAT_GRAYRESPONSE / MSG_SET

Description

This structure is used by the application to specify a set of mapping values to be applied to grayscale data. Use this structure for grayscale data whose bit depth is up to and including 8-bits. This structure can only be used if TW_IMAGEINFO.PixelType is TWPT_GRAY. The number of elements in the array is determined by TW_IMAGEINFO.BitsPerPixel—the number of elements is 2 raised to the power of TW_IMAGEINFO.BitsPerPixel.

This structure is primarily intended for use by applications that bypass the Source's built-in user interface.

Field Descriptions

Response[1]	Transfer curve descriptors. All three channels must contain the same value for every entry.

TW_HANDLE

On Windows:

```
typedef HANDLE          TW_HANDLE;
```

On Macintosh:

```
typedef Handle          TW_HANDLE;
```

On Unix:

```
typedef unsigned char   *TW_HANDLE;
```

Used by

Embedded in the TW_CAPABILITY and TW_USERINTERFACE structures

Description

The typedef of Handles are defined by the operating system. TWAIN defines TW_HANDLE to be the handle type supported by the operating system.

Field Descriptions

See definitions above

TW_IDENTITY

```
typedef struct {
   TW_UINT32       Id;
   TW_VERSION       Version;
   TW_UINT16       ProtocolMajor;
   TW_UINT16       ProtocolMinor;
   TW_UINT32       SupportedGroups;
   TW_STR32        Manufacturer;
   TW_STR32        ProductFamily;
   TW_STR32        ProductName;
} TW_IDENTITY, FAR * pTW_IDENTITY;
```

Used by

A large number of the operations because it identifies the application and the Source

Description

Provides identification information about a TWAIN entity. Used to maintain consistent communication between entities.

Field Descriptions

Id	A unique, internal identifier for the TWAIN entity. This field is only filled by the Source Manager. Neither an application nor a Source should fill this field. The Source uses the contents of this field to "identify" which application is invoking the operation sent to the Source.
Version	A TW_VERSION structure identifying the TWAIN entity.
ProtocolMajor	Major number of latest TWAIN version that this element supports (see TWON_PROTOCOLMAJOR).
ProtocolMinor	Minor number of latest TWAIN version that this element supports (see TWON_PROTOCOLMINOR).
SupportedGroups	1. The application will normally set this field to specify which Data Group(s) it wants the Source Manager to sort Sources by when presenting the Select Source dialog, or returning a list of available Sources. The application sets this prior to invoking a MSG_USERSELECT operation. 2. The application may also set this field to specify which Data Group(s) it wants the Source to be able to acquire and transfer. The application must do this prior to sending the Source its MSG_ENABLEDS operation. 3. The Source must set this field to specify which Data Group(s) it can acquire. It will do this in response to a MSG_OPENDS.
Manufacturer	String identifying the manufacturer of the application or Source. e.g. "Aldus".
ProductFamily	Tells an application that performs device-specific operations which product family the Source supports. This is useful when a new Source has been released and the application doesn't know about the particular Source but still wants to perform Custom operations with it. e.g. "ScanMan".
ProductName	A string uniquely identifying the Source. This is the string that will be displayed to the user at Source select-time. This string must uniquely identify your Source for the user, and should identify the application unambiguously for Sources that care. e.g. "ScanJet IIc".

TW_IMAGEINFO

```
typedef struct {
   TW_FIX32       XResolution;
   TW_FIX32       YResolution;
   TW_INT32       ImageWidth;
   TW_INT32       ImageLength;
   TW_INT16       SamplesPerPixel;
   TW_INT16       BitsPerSample[8];
   TW_INT16       BitsPerPixel;
   TW_BOOL        Planar;
   TW_INT16       PixelType;
   TW_UINT16      Compression;
} TW_IMAGEINFO, FAR * pTW_IMAGEINFO;
```

Used by

The DG_IMAGE / DAT_IMAGEINFO / MSG_GET operation

Description

Describes the "real" image data, that is, the complete image being transferred between the Source and application. The Source may transfer the data in a different format--the information may be transferred in "strips" or "tiles" in either compressed or uncompressed form. See the TW_IMAGEMEMXFER structure for more information.

The term "sample" is referred to a number of times in this structure. It holds the same meaning as in the TIFF specification. A sample is a contiguous body of image data that can be categorized by the channel or "ink color" it was captured to describe. In an R-G-B (Red-Green-Blue) image, such as on your TV or computer's CRT, each color channel is composed of a specific color. There are 3 samples in an R-G-B; Red, Green, and Blue. A C-Y-M-K image has 4 samples. A Grayscale or Black and White image has a single sample.

Note: The value -1 in ImageWidth and ImageLength are special cases. It is possible for a Source to not know either its Width or Length. Applications need to consider this when allocating memory or otherwise dealing with the size of the Image.

Field Descriptions

XResolution	The number of pixels per ICAP_UNITS in the horizontal direction. The current unit is assumed to be "inches" unless it has been otherwise negotiated between the application and Source.
YResolution	The number of pixels per ICAP_UNITS in the vertical direction.
ImageWidth	How wide, in pixels, the entire image to be transferred is. If the Source doesn't know, set this field to -1 (hand scanners may do this). --1 can only be used if the application has set ICAP_UNDEFINEDIMAGESIZE to TRUE.
ImageLength	How tall/long, in pixels, the image to be transferred is. If the Source doesn't know, set this field to -1 (hand scanners may do this). -1 can only be used if the application has set ICAP_UNDEFINEDIMAGESIZE to TRUE.
SamplesPerPixel	The number of samples being returned. For R-G-B, this field would be set to 3. For C-M-Y-K, 4. For Grayscale or Black and White, 1.
BitsPerSample[8]	For each sample, the number of bits of information. 24-bit R-G-B will typically have 8 bits of information in each sample (8+8+8). Some 8-bit color is sampled at 3 bits Red, 3 bits Green, and 2 bits Blue. Such a scheme would put 3, 3, and 2 into the first 3 elements of this array. The supplied array allows up to 8 samples. Samples are not limited to 8 bits. However, both the application and Source must simultaneously support sample sizes greater than 8 bits per color.
BitsPerPixel	The number of bits in each image pixel (or bit depth). This value is invariant across the image. 24-bit R-G-B has BitsPerPixel = 24. 40-bit C-M-Y-K has BitsPerPixel=40. 8-bit Grayscale has BitsPerPixel = 8. Black and White has BitsPerPixel = 1.
Planar	If SamplesPerPixel > 1, indicates whether the samples follow one another on a pixel-by-pixel basis (R-G-B-R-G-B-R-G-B...) as is common with a one-pass scanner or all the pixels for each sample are grouped together (complete group of R, complete group of G, complete group of B) as is common with a three-pass scanner. If the pixel-by-pixel method (also known as "chunky") is used, the Source should set Planar = FALSE. If the grouped method (also called "planar") is used, the Source should set Planar = TRUE.
PixelType	This is the highest categorization for how the data being transferred should be interpreted by the application. This is how the application can tell if the data is Black and White, Grayscale, or Color. Currently, the only color type defined is "tri-stimulus", or color described by three characteristics. Most popular color description methods use tri-stimulus descriptors. For simplicity, the constant used to identify tri-stimulus color is called TWPT_RBG, for R-G-B color. There is no default for this value. Fill this field with the appropriate TWPT_xxxx constant.
Compression	The compression method used to process the data being transferred. Default is no compression. Fill this field with the appropriate TWCP_xxxx constant.

TW_IMAGELAYOUT

```
typedef struct {
   TW_FRAME      Frame;
   TW_UINT32      DocumentNumber;
   TW_UINT32      PageNumber;
   TW_UINT32      FrameNumber;
} TW_IMAGELAYOUT, FAR * pTW_IMAGELAYOUT;
```

Used by

DG_IMAGE / DAT_IMAGELAYOUT / MSG_GET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_GETDEFAULT
DG_IMAGE / DAT_IMAGELAYOUT / MSG_RESET
DG_IMAGE / DAT_IMAGELAYOUT / MSG_SET

Description

Involves information about the original size of the acquired image and its position on the original "page" relative to the "page's" upper-left corner. **Default measurements are in inches** (units of measure can be changed by negotiating the ICAP_UNITS capability). This information may be used by the application to relate the acquired (and perhaps processed image) to the original. Further, the application can, using this structure, set the size of the image it wants acquired.

Another attribute of this structure is the included frame, page, and document indexing information. Most Sources and applications, at least at first, will likely set all these fields to one. For Sources that can acquire more than one frame from a page in a single acquisition, the FrameNumber field will be handy. Sources that can acquire more than one page from a document feeder will use PageNumber and DocumentNumber. These fields will be especially useful for forms-processing applications and other applications with similar document tracking requirements.

Field Descriptions

Frame	Defines the Left, Top, Right, and Bottom coordinates (in ICAP_UNITS) of the rectangle enclosing the original image on the original "page". If the application isn't interested in setting the origin of the image, set both Top and Left to zero. The Source will fill in the actual values following the acquisition. See also TW_FRAME.
DocumentNumber	The document number, assigned by the Source, that the acquired data originated on. Useful for grouping pages together. Usually a physical representation, this could just as well be a logical construct. Initial value is 1. Increment when a new document is placed into the document feeder (usually tell this has happened when the feeder empties). Reset when no longer acquiring from the feeder.
PageNumber	The page which the acquired data was captured from. Useful for grouping Frames together that are in some way related, usually Source. Usually a physical representation, this could just as well be a logical construct. Initial value is 1. Increment for each page fed from a page feeder. Reset when a new document is placed into the feeder.
FrameNumber	Usually a chronological index of the acquired frame. These frames are related to one another in some way; usually they were acquired from the same page. The Source assigns these values. Initial value is 1. Reset when a new page is acquired from.

TW_IMAGEMEMXFER

```
typedef struct {
   TW_UINT16       Compression;
   TW_UINT32       BytesPerRow;
   TW_UINT32       Columns;
   TW_UINT32       Rows;
   TW_UINT32       XOffset;
   TW_UINT32       YOffset;
   TW_UINT32       BytesWritten;
   TW_MEMORY       Memory;
} TW_IMAGEMEMXFER, FAR * pTW_IMAGEMEMXFER;
```

Used by

DG_IMAGE / DAT_IMAGEMEMXFER / MSG_GET

Description

Describes the form of the acquired data being passed from the Source to the application. When used in combination with a TW_IMAGEINFO structure, the application can correctly interpret the image.

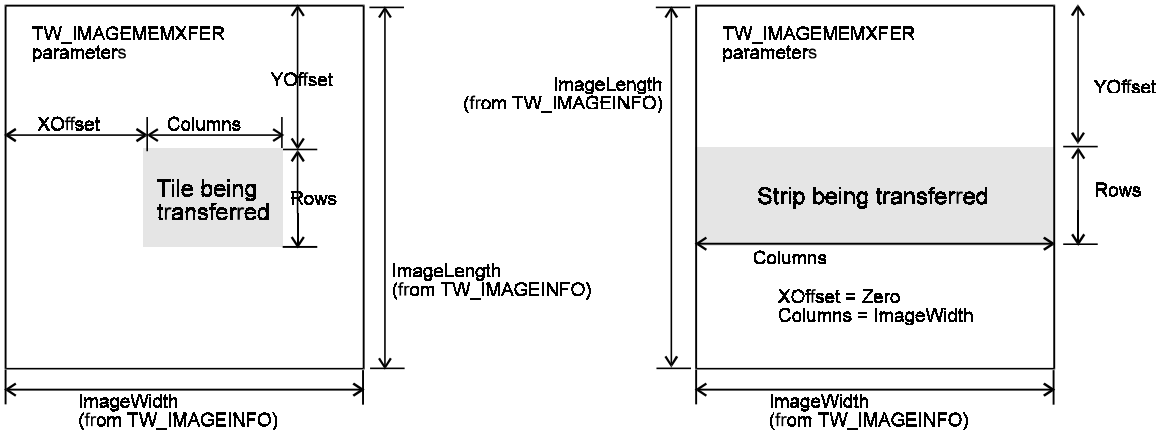
This structure allows transfer of "chunks" from the acquired data. These portions may be either "strips" or "tiles." Strips are tiles whose width matches that of the full image. Strips are always passed sequentially, from "top" to "bottom". A tile's position does not necessarily follow that of the previously passed tile. Most Sources will transfer strips.

Note: The application should remember what corner was contained in the first tile of a plane. When the opposite corner is delivered, the plane is complete. The dimensions of the memory transfers may vary.

Data may be passed either compressed or uncompressed. All Sources must pass uncompressed data. Sources are not required to support compressed data transfers. Compressed data transfers, and how the values are entered into the fields of this structure, are described in Chapter 4.

Following is a picture of some of the fields from a TW_IMAGEMEMXFER structure. **The large outline shows the entire image which was selected to be transferred.** The smaller rectangle shows the particular portion being described by this TW_IMAGEMEMXFER structure.

Note: Remember that for a "strip" transfer XOffset = 0, and Columns = TW_IMAGEINFO.ImageWidth.



Field Descriptions

Compression	The compression method used to process the data being transferred. Write the constant (TWCP_xxxx) that precisely describes the type of compression used for the buffer. This may be different from the method reported in the TW_IMAGEINFO structure (if the user selected a different method before the actual transfer began, for instance). This is unlikely, but possible. The application can optionally abort the acquisition if the value in this field differs from the TW_IMAGEINFO value. Default is no compression (TWCP_NONE) and most transfers will probably be uncompressed. See the list of constants in the TWAIN.H file.
BytesPerRow	The number of uncompressed bytes in each row of the piece of the image being described in this buffer.
Columns	The number of uncompressed columns (in pixels) in this buffer.
Rows	The number or uncompressed rows (in pixels) in this buffer.
XOffset	How far, in pixels, the left edge of the piece of the image being described by this structure is inset from the "left" side of the original image. If the Source is transferring in "strips", this value will equal zero. If the Source is transferring in "tiles", this value will often be non-zero.
YOffset	Same idea as XOffset, but the measure is in pixels from the "top" of the original image to the upper edge of this piece.
BytesWritten	The number of bytes written into the transfer buffer. This field must always be filled in correctly, whether compressed or uncompressed data is being transferred.
Memory	A structure of type TW_MEMORY describing who must dispose of the buffer, the actual size of the buffer, in bytes, and where the buffer is located in memory.

TW_JPEGCOMPRESSION

```
typedef struct {
   TW_UINT16       ColorSpace;
   TW_UINT32       SubSampling;
   TW_UINT16       NumComponents;
   TW_UINT16       RestartFrequency;
   TW_UINT16       QuantMap[4]Manufacturer;
   TW_MEMORY      QuantTable[4];
   TW_UINT16       HuffmanMap[4];
   TW_MEMORY       HuffmanDC[2];
   TW_MEMORY       HuffmanAC[2];
} TW_JPEGCOMPRESSION, FAR * pTW_JPEGCOMPRESSION;
```

Used by

DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GET
DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_GETDEFAULT
DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_RESET
DG_IMAGE / DAT_JPEGCOMPRESSION / MSG_SET

Description

Describes the information necessary to transfer a JPEG-compressed image during a buffered transfer. Images compressed in this fashion will be compatible with the JPEG File Interchange Format, version 1.1. For more information on JPEG and TWAIN, see Chapter 4. The TWAIN JPEG implementation is based on the JPEG Draft International Standard, version 10918-1. The sample tables found in Section K of the JPEG Draft International Standard, version 10918-1 are used as the default tables for QuantTable, HuffmanDC, and HuffmanAC.

Field Descriptions

ColorSpace	One of the TWPT_xxxx values. Defines the color space in which the compressed components are stored. Only spaces supported by the Source for ICAP_JPEGPIXELTYPE are valid.
SubSampling	Encodes the horizontal and vertical subsampling in the form ABCDEFGH, where ABCD are the high-order four nibbles which represent the horizontal subsampling and EFGH are the low-order four nibbles which represent the vertical subsampling. Each nibble may have a value of 0, 1, 2, 3, or 4. However, max(A,B,C,D) * max(E,F,G,H) must be less than or equal to 10. Subsampling is irrelevant for single component images. Therefore, the corresponding nibbles should be set to 1. e.g. To indicate subsampling two Y for each U and V in a YUV space image, where the same subsampling occurs in both horizontal and vertical axes, this field would hold 0x21102110. For a grayscale image, this field would hold 0x10001000. A CMYK image could hold 0x11111111.
NumComponents	Number of color components in the image to be compressed.
RestartFrequency	Number of MDUs (Minimum Data Units) between restart markers. Default is 0, indicating that no restart markers are used. An MDU is defined for interleaved data (i.e. R-G-B, Y-U-V, etc.) as a minimum complete set of 8x8 component blocks.
QuantMap[4]	Mapping of components to Quantization tables.
QuantTable[4]	Quantization tables.
HuffmanMap[4]	Mapping of components to Huffman tables. Null entries signify selection of the default tables.
HuffmanDC[2]	DC Huffman tables. Null entries signify selection of the default tables.
HuffmanAC[2]	AC Huffman tables. Null entries signify selection of the default tables.

TW_MEMORY

```
typedef struct {
   TW_UINT32      Flags;
   TW_UINT32      Length;
   TW_MEMREF      TheMem;
} TW_MEMORY, FAR * pTW_MEMORY;
```

Used by

Embedded in the TW_IMAGEMEMXFER and TW_JPEGCOMPRESSION structures

Description

Provides information for managing memory buffers. Memory for transfer buffers is allocated by the application--the Source is asked to fill these buffers. This structure keeps straight which entity is responsible for deallocation.

Field Descriptions

Flags — Encodes which entity releases the buffer and how the buffer is referenced. The ownership flags must be used:

- when transferring Buffered Memory data as tiles
- when transferring Buffered Memory that is compressed
- in the TW_JPEGCOMPRESSION structure

When transferring Buffered Memory data as uncompressed strips, the application allocates the buffers and is responsible for setting the ownership flags.

This field is used to identify how the memory is to be referenced. The memory is always referenced by a Handle on the Macintosh and a Pointer under UNIX. It is referenced by a Handle or a pointer under MS Windows.

Use TWMF_xxxx constants, bit-wise OR'd together to fill this field.

Flag Constants:

TWMF_APPOWNS	0x1
TWMF_DSMOWNS	0x2
TWMF_DSOWNS	0x4
TWMF_POINTER	0x8
TWMF_HANDLE	0x10

Length — The size of the buffer in bytes. Should always be an even number and word-aligned.

TheMem — Reference to the buffer. May be a Pointer or a Handle (see Flags field to make this determination). You must typecast this field before referencing it in your code.

TW_MEMREF

On Windows:

```
typedef LPVOID     TW_MEMREF;
```

On Macintosh:

```
typedef char       *TW_MEMREF;
```

On Unix:

```
typedef unsigned charEMREF;
```

Used by

Embedded in the TW_EVENT and TW_MEMORY structures

Description

Memory references are specific to each operating system. TWAIN defines TW_MEMREF to be the memory reference type supported by the operating system.

Field Descriptions

See definitions above

TW_ONEVALUE

```
typedef struct {
   TW_UINT16      ItemType;
   TW_UINT32      Item;
} TW_ONEVALUE, FAR * pTW_ONEVALUE;
```

Used by

TW_CAPABILITY structure (when ConType field specifies TWON_ONEVALUE)

Description

Stores a single value (item) which describes a capability. This structure is currently used only in a TW_CAPABILITY structure. Such a value would be useful to describe the current value of the device's contrast, or to set a specific contrast value. This structure is related in function and purpose to TW_ARRAY, TW_ENUMERATION, and TW_RANGE.

Note that in cases where the data type is TW_UINT16, the data should reside in the lower word.

Field Descriptions

ItemType	The type of the item. The type is indicated by the constant held in this field. The constant is of the kind TWTY_xxxx.
Item	The value.

TW_PALETTE8

```
typedef struct {
   TW_UINT16      NumColors;
   TW_UINT16      PaletteType;
   TW_ELEMENT8    Colors[256];
} TW_PALETTE8, FAR * pTW_PALETTE8;
```

Used by

DG_IMAGE / DAT_PALETTE8 / MSG_GET
DG_IMAGE / DAT_PALETTE8 / MSG_GETDEFAULT
DG_IMAGE / DAT_PALETTE8 / MSG_RESET
DG_IMAGE / DAT_PALETTE8 / MSG_SET

Description

This structure holds the color palette information for buffered memory transfers of type ICAP_PIXELTYPE = TWPT_PALETTE.

Field Descriptions

NumColors	Number of colors in the color table; maximum index into the color table should be one less than this (since color table indexes are zero-based).
PaletteType	TWPA_xxxx constant specifying the type of palette.
Colors[256]	Array of palette values.

TW_PENDINGXFERS

```
typedef struct {
   TW_UINT16      Count;
   TW_UINT32      Reserved;
} TW_PENDINGXFERS, FAR * pTW_PENDINGXFERS;
```

Used by

DG_CONTROL / DAT_PENDINGXFERS / MSG_ENDXFER
DG_CONTROL / DAT_PENDINGXFERS / MSG_GET
DG_CONTROL / DAT_PENDINGXFERS / MSG_RESET

Description

This structure tells the application how many more complete transfers the Source currently has available. The application should MSG_GET this structure at the conclusion of a transfer to confirm the Source's current state. If the Source has more transfers pending it will remain in State 6 awaiting initiation of the next transfer by the application. If it has no more transfers pending, it will place zero into Count and will have automatically transitioned to State 5. If the Source knows there are more transfers pending but is unsure of the actual number, it should place -1 into Count (for example, with document feeders or continuous video sources). Otherwise, the Source should place the actual number of pending transfers into Count.

Field Descriptions

Count	The number of complete transfers a Source has available for the application it is connected to. If no more transfers are available, set to zero. If an unknown and non-zero number of transfers are available, set to -1.
Reserved	Reserved for future use.

TW_RANGE

```
typedef struct {
   TW_UINT16      ItemType;
   TW_UINT32      MinValue;
   TW_UINT32      MaxValue;
   TW_UINT32      StepSize;
   TW_UINT32      DefaultValue;
   TW_UINT32      CurrentValue;
} TW_RANGE, FAR * pTW_RANGE;
```

Used by

TW_CAPABILITY structure (when ConType field specifies TWON_RANGE)

Description

Stores a range of individual values describing a capability. The values are uniformly distributed between a minimum and a maximum value. The step size between each value is constant. Such a value is useful when describing such capabilities as the resolutions of a device which supports discreet, uniform steps between each value, such as 50 through 300 dots per inch in steps of 2 dots per inch (50, 52, 54, ..., 296, 298, 300). This structure is related in function and purpose to TW_ARRAY, TW_ENUMERATION, and TW_ONEVALUE.

Field Descriptions

ItemType	The type of items in the list. The type is indicated by the constant held in this field. The constant is of the kind TWTY_xxxx. All items in the list have the same size/type.
MinValue	The least positive/most negative value of the range.
MaxValue	The most positive/least negative value of the range.
StepSize	The delta between two adjacent values of the range. e.g. Item2 - Item1 = StepSize;
DefaultValue	The device's "power-on" value for the capability. If the application is performing a MSG_SET operation and isn't sure what the default value is, set this field to TWON_DONTCARE32.
CurrentValue	The value to which the device (or its user interface) is currently set to for the capability.

TW_RGBRESPONSE

```
typedef struct {
   ELEMENT8      Response[1];
} TW_RGBRESPONSE, FAR * pTW_RGBRESPONSE;
```

Used by

DG_IMAGE / DAT_RGBRESPONSE / MSG_RESET
DG_IMAGE / DAT_RGBRESPONSE / MSG_SET

Description

This structure is used by the application to specify a set of mapping values to be applied to RGB color data. Use this structure for RGB data whose bit depth is up to, and including, 8-bits. The number of elements in the array is determined by TW_IMAGEINFO.BitsPerPixel—the number of elements is 2 raised to the power of TW_IMAGEINFO.BitsPerPixel.

This structure is primarily intended for use by applications that bypass the Source's built-in user interface.

Field Descriptions

Response[1]	Transfer curve descriptors. To minimize color shift problems, writing the same values into each channel is desirable.

TW_SETUPFILEXFER

```
typedef struct {
   TW_STR255      FileName;
   TW_UINT16      Format;
   TW_INT16      VRefNum;
} TW_SETUPFILEXFER, FAR * pTW_SETUPFILEXFER;
```

Used by

DG_CONTROL / DAT_SETUPFILEXFER / MSG_GET
DG_CONTROL / DAT_SETUPFILEXFER / MSG_GETDEFAULT
DG_CONTROL / DAT_SETUPFILEXFER / MSG_RESET
DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET

Description

Describes the file format and file specification information for a transfer through a disk file.

Field Descriptions

FileName	A complete file specifier to the target file. On Windows, be sure to include the complete pathname.
Format	The format of the file the Source is to fill. Fill with the correct constant—as negotiated with the Source—of type TWFF_xxxx.
VRefNum	The volume reference number for the file. This applies to Macintosh only. On Windows, fill the field with TWON_DONTCARE16.

TW_SETUPMEMXFER

```
typedef struct {
   TW_UINT32      MinBufSize;
   TW_UINT32      MaxBufSize;
   TW_UINT32      Preferred;
} TW_SETUPMEMXFER, FAR * pTW_SETUPMEMXFER;
```

Used by

DG_CONTROL / DAT_SETUPMEMXFER / MSG_GET

Description

Provides the application information about the Source's requirements and preferences regarding allocation of transfer buffer(s). The best applications will allocate buffers of the Preferred size. An application should never allocate a buffer smaller than MinBufSize. Some Sources may not be able to fill a buffer larger than MaxBufSize so a larger allocation is a waste of RAM (digital cameras or frame grabbers fit this category).

Sources should fill out all three fields as accurately as possible. If a Source can fill an indeterminately large buffer (hand scanners might do this), put a -1 in MaxBufSize.

Field Descriptions

MinBufSize	The size of the smallest transfer buffer, in bytes, that a Source can be successful with. This will typically be the number of bytes in an uncompressed row in the block to be transferred. An application should never allocate a buffer smaller than this.
MaxBufSize	The size of the largest transfer buffer, in bytes, that a Source can fill. If a Source can fill an arbitrarily large buffer, it might set this field to negative 1 to indicate this (a hand-held scanner might do this, depending on how long its cord is). Other Sources, such as frame grabbers, cannot fill a buffer larger than a certain size. Allocation of a transfer buffer larger than this value is wasteful.
Preferred	The size of the optimum transfer buffer, in bytes. A smart application will allocate transfer buffers of this size, if possible. Buffers of this size will optimize the Source's performance. Sources should be careful to put reasonable values in this field. Buffers that are 10's of kbytes will be easier for applications to allocate than buffers that are 100's or 1000's of kbytes.

TW_STATUS

```
typedef struct {
   TW_UINT16      ConditionCode;
   TW_UINT16      Reserved;
} TW_STATUS, FAR * pTW_STATUS;
```

Used by

DG_CONTROL / DAT_STATUS / MSG_GET

Description

Used to describe the status of a Source. To ask the Source to fill in this structure, the application sends:

DG_CONTROL / DAT_STATUS / MSG_GET

with a pointer to a TW_STATUS structure. This is typically done in response to a Return Code other than TWRC_SUCCESS and should always be done in response to a Return Code of TWRC_CHECKSTATUS. In such a case, the Source has something it needs the application to know about.

Field Descriptions

ConditionCode	The TWCC_xxxx code (Condition Code) being returned to the application.
Reserved	Reserved for future use.

TW_TRANSFORMSTAGE

```
typedef struct {
   TW_DECODEFUNCTION      Decode[3];
   TW_FIX32               Mix[3][3];
} TW_TRANSFORMSTAGE, FAR * pTW_TRANSFORMSTAGE;
```

Used by

Embedded in the TW_CIECOLOR structure

Description

Specifies the parametrics used for either the ABC or LMN transform stages. This structure parallels the TTransformStage structure definition in Appendix A.

Field Descriptions

Decode[3]	Channel-specific transform parameters.
Mix[3][3]	3x3 matrix that specifies how channels are mixed in

TW_USERINTERFACE

```
typedef struct {
   TW_BOOL        ShowUI;
   TW_BOOL        ModalUI;
   TW_HANDLE      hParent;
} TW_USERINTERFACE, FAR * pTW_USERINTERFACE;
```

Used by

DG_CONTROL / DAT_USERINTERFACE / MSG_DISABLEDS
DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

Description

This structure is used to handle the user interface coordination between an application and a Source.

Field Descriptions

ShowUI	Set to TRUE by the application if the Source should activate its built-in user interface. Otherwise, set to FALSE. Note that not all sources support ShowUI = FALSE. See the description of DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS for more information.
ModalUI	Set to TRUE by the Source if the Source's built-in user interface behaves modally. This is described in Chapter 4 of this Toolkit.
hParent	MS Windows only: Application's window handle. The Source designates the hWnd as its parent when creating the Source dialog. **NOTE:** Window handle allows Source's user interface to be a proper child of the parent application.

TW_VERSION

```
typedef struct {
   TW_UINT16      MajorNum;
   TW_UINT16      MinorNum;
   TW_UINT16      Language;
   TW_UINT16      Country;
   TW_STR32      Info;
} TW_VERSION, FAR * pTW_VERSION;
```

Used by

This is embedded in the TW_IDENTITY data structure

Description

A general way to describe the version of software that is running.

Field Descriptions

MajorNum	This refers to your application or Source's major revision number. e.g. The "2" in "version 2.01".
MinorNum	The incremental revision number of your application or Source. e.g. The "1" in "version 2.1".
Language	The primary language for your Source or application. e.g. TWLG_GER.
Country	The primary country where your Source or application is intended to be distributed. e.g. Germany.
Info	General information string - fill in as needed. e.g. "1.0b3 Beta release".

Data Argument Types that Don't Have Associated TW_Structures

Most of the DAT_xxxx components of the TWAIN operation triplets have a corresponding data structure whose name begins with TW_ and then uses the same suffix as the DAT_ name. However, the following do not use that pattern.

DAT_IMAGEFILEXFER

Acts on NULL data.

DAT_IMAGENATIVEXFER

Uses a TW_UINT32 variable.

- **On Windows:** In Win 3.1, the low word of this 32-bit integer is a handle variable to a DIB (Device Independent Bitmap) located in memory. For Win 95 the handles fill the entire field.
- **On Macintosh:** This 32-bit integer is a handle to a Picture (a PicHandle). It is a QuickDraw picture located in memory.

DAT_NULL

Used by the Source to signal the need for an event to announce MSG_XFERREADY or MSG_CLOSEDSREQ. (Used on Windows only)

DAT_PARENT

Used by the DG_CONTROL / DAT_PARENT / MSG_OPENDSM and MSG_CLOSEDSM operations.

- **On Windows:** They act on a variable of type TW_INT32. Prior to the operation, the application must write, a window handle to the application's window that acts as the "parent" for the Source's user interface. In Win 3.1 this would be in the low word, in Win 95 it will fill the entire field. (This must be done whether or not the Source's user interface will be used. The Source Manager uses this window handle to signal the application when data is ready for transfer (MSG_XFERREADY) or the Source needs to be closed (MSG_CLOSEDSREQ)).
- **On Macintosh:** These act on NULL data.

DAT_XFERGROUP

Used by the DG_CONTROL / DAT_XFERGROUP / MSG_GET operation. The data acted on by this operation is a variable of type TW_UINT32. (The same as a DG_xxxx designator.) The value of this variable is indeterminate prior to the operation. Following the operation, a single bit is set indicating the Data Group of the transfer.

Constants

Generic Constants

Constants		(defined as)
	TWON_PROTOCOLMINOR	0
	TWON_PROTOCOLMAJOR	1
	TWON_ARRAY	3
	TWON_ENUMERATION	4
	TWON_ONEVALUE	5
	TWON_RANGE	6
	TWON_ICONID	962
	TWON_DSMID	461
	TWON_DSMCODEID	63
	TWON_DONTCARE8	0xff
	TWON_DONTCARE16	0xffff
	TWON_DONTCARE32	0xffffffff
Flags used in TW_MEMORY		
	TWMF_APPOWNS	0x1
	TWMF_DSMOWNS	0x2
	TWMF_DSOWNS	0x4
	TWMF_POINTER	0x8
	TWMF_HANDLE	0x10
Palette types for TW_PALETTE8		
	TWPA_RGB	0
	TWPA_GRAY	1
	TWPA_CMY	2
ItemTypes for Capability Container structures		
	TWTY_INT8	0x0000
	TWTY_INT16	0x0001
	TWTY_INT32	0x0002
	TWTY_UINT8	0x0003
	TWTY_UINT16	0x0004
	TWTY_UINT32	0x0005
	TWTY_BOOL	0x0006
	TWTY_FIX32	0x0007
	TWTY_FRAME	0x0008
	TWTY_STR32	0x0009
	TWTY_STR64	0x000a
	TWTY_STR128	0x000b
	TWTY_STR255	0x000c

Capability Constants

ICAP_BITDEPTHREDUCTION values		(defined as)
	TWBR_THRESHOLD	0
	TWBR_HALFTONES	1
	TWBR_CUSTHALFTONE	2
	TWBR_DIFFUSION	3
ICAP_BITORDER values		
	TWBO_LSBFIRST	0
	TWBO_MSBFIRST	1
ICAP_COMPRESSION values		
	TWCP_NONE	0
	TWCP_PACKBITS	1
	TWCP_GROUP31D	2
	TWCP_GROUP31DEOL	3
	TWCP_GROUP32D	4
	TWCP_GROUP34	5
	TWCP_JPEG	6
	TWCP_LZW	7
ICAP_IMAGEFILEFORMAT values		
	TWFF_TIFF	0
	TWFF_PICT	1
	TWFF_BMP	2
	TWFF_XBM	3
	TWFF_JFIF	4
ICAP_FILTER values		
	TWFT_RED	0
	TWFT_GREEN	1
	TWFT_BLUE	2
	TWFT_NONE	3
	TWFT_WHITE	4
	TWFT_CYAN	5
	TWFT_MAGENTA	6
	TWFT_YELLOW	7
	TWFT_BLACK	8
ICAP_LIGHTPATH values		
	TWLP_REFLECTIVE	0
	TWLP_TRANSMISSIVE	1
ICAP_LIGHTSOURCE values		
	TWLS_RED	0
	TWLS_GREEN	1
	TWLS_BLUE	2
	TWLS_NONE	3
	TWLS_WHITE	4
	TWLS_UV	5
	TWLS_IR	6

ICAP_ORIENTATION values

TWOR_ROT0	0
TWOR_ROT90	1
TWOR_ROT180	2
TWOR_ROT270	3
TWOR_PORTRAIT	TWOR_ROT0
TWOR_LANDSCAPE	TWOR_ROT270

ICAP_PLANARCHUNKY values

TWPC_CHUNKY	0
TWPC_PLANAR	1

ICAP_PIXELFLAVOR values

TWPF_CHOCOLATE	0
TWPF_VANILLA	1

ICAP_PIXELTYPE values

TWPT_BW	0
TWPT_GRAY	1
TWPT_RGB	2
TWPT_PALETTE	3
TWPT_CMY	4
TWPT_CMYK	5
TWPT_YUV	6
TWPT_YUVK	7
TWPT_CIEXYZ	8

ICAP_SUPPORTEDSIZES values

TWSS_NONE	0
TWSS_A4LETTER	1
TWSS_B5LETTER	2
TWSS_USLETTER	3
TWSS_USLEGAL	4
TWSS_A5	5
TWSS_B4	6
TWSS_B6	7
TWSS_B	8

ICAP_XFERMECH values

TWSX_NATIVE	0
TWSX_FILE	1
TWSX_MEMORY	2

ICAP_UNITS values

TWUN_INCHES	0
TWUN_CENTIMETERS	1
TWUN_PICAS	2
TWUN_POINTS	3
TWUN_TWIPS	4
TWUN_PIXELS	5

Language Constants

Language		(defined as)
Danish	TWLG_DAN	0
Dutch	TWLG_DUT	1
International English	TWLG_ENG	2
French Canadian	TWLG_FCF	3
Finnish	TWLG_FIN	4
French	TWLG_FRN	5
German	TWLG_GER	6
Icelandic	TWLG_ICE	7
Italian	TWLG_ITN	8
Norwegian	TWLG_NOR	9
Portuguese	TWLG_POR	10
Spanish	TWLG_SPA	11
Swedish	TWLG_SWE	12
U.S. English	TWLG_USA	13

9

Capabilities

Chapter Contents

Overview

Sources may support a large number of capabilities but are required to support very few. To determine if a capability is supported by a Source, the application can query the Source using a DG_CONTROL / DAT_CAPABILITY / MSG_GET, MSG_GETCURRENT, or MSG_GETDEFAULT operation. The application specifies the particular capability by storing its identifier in the Cap field of the TW_CAPABILITY structure. This is the structure pointed to by the pData parameter in the DSM_Entry( ) call.

DG_CONTROL / DAT_CAPABILITY operations for capability negotiation include:

MSG_GET	Returns the available settings for this capability, as well as current and default settings (if container is TW_ENUMERATION or TW_RANGE).
MSG_GETCURRENT	Returns the current setting for this capability.
MSG_GETDEFAULT	Returns the value of the Source's preferred default values.
MSG_RESET	Returns the capability to its TWAIN default (power-on) condition (i.e. all previous negotiation is ignored).
MSG_SET	Allows the application to set the current value of a capability or even to restrict the available values to some subset of the Source's power-on set of values. Sources are strongly encouraged to allow the application to set as many of its capabilities as possible, and further to reflect these changes in the Source's user interface. This will ensure that the user can only select images with characteristics that are useful to the consuming application.

Required Capabilities

The list of required capabilities can be found in Chapter 5.

Sources must implement and make available to TWAIN applications the advertised features of the devices they support. This is especially true in no UI mode. Thus, when a capability is listed as required by none, a Source must still support it if its device supports it.

Capabilities in Categories of Functionality

Capability Negotiation parameters

CAP_EXTENDEDCAPS	Capabilities negotiated in States 5 & 6
CAP_SUPPORTEDCAPS	Inquire Source's capabilities valid for MSG_GET

Color

ICAP_FILTER	Color characteristics of the subtractive filter applied to the image data
ICAP_GAMMA	Gamma correction value for the image data
ICAP_PLANARCHUNKY	Color data format - Planar or Chunky

Compression

ICAP_BITORDERCODES	CCITT Compression
ICAP_CCITTKFACTOR	CCITT Compression
ICAP_COMPRESSION	Compression method for Buffered Memory Transfers
ICAP_JPEGPIXELTYPE	JPEG Compression
ICAP_PIXELFLAVORCODES	CCITT Compression
ICAP_TIMEFILL	CCITT Compression

Device Parameters

ICAP_PHYSICALHEIGHT	Maximum height Source can acquire (in ICAP_UNITS)
ICAP_PHYSICALWIDTH	Maximum width Source can acquire (in ICAP_UNITS)
ICAP_UNITS	Unit of measure (inches, centimeters, etc.)
ICAP_EXPOSURETIME	Exposure time used to capture the image, in seconds
ICAP_FLASHUSED	Was a flash used to acquire image?
ICAP_LAMPSTATE	Is the lamp on?
ICAP_LIGHTPATH	Image was captured transmissively or reflectively
ICAP_LIGHTSOURCE	Describes the color characteristic of the light source used to acquire the image
CAP_DEVICEONLINE	Determines if hardware is on and ready

Using a Feeder

CAP_AUTOFEED	MSG_SET to TRUE to enable Source's automatic feeding
CAP_CLEARPAGE	MSG_SET to TRUE to eject current page and leave acquire area empty
CAP_FEEDERENABLED	If TRUE, Source's feeder is available
CAP_FEEDERLOADED	If TRUE, Source has documents loaded in feeder (MSG_GET only)
CAP_FEEDPAGE	MSG_SET to TRUE to eject current page and feed next page
CAP_REWINDPAGE	MSG_SET to TRUE to do a reverse feed

Image Information

CAP_AUTHOR	Author of acquired image (may include a copyright string)
CAP_CAPTION	General note about acquired image
CAP_TIMEDATE	Date and Time the image was acquired (entered State 7)

Image Parameters for Acquire

ICAP_AUTOBRIGHT	Enable Source's Auto-brightness function
ICAP_BRIGHTNESS	Source brightness values
ICAP_CONTRAST	Source contrast values
ICAP_HIGHLIGHT	Lightest highlight, values lighter than this value will be set to this value
ICAP_ORIENTATION	Defines which edge of the paper is the top: Portrait or Landscape
ICAP_ROTATION	Source can, or should, rotate image this number of degrees
ICAP_SHADOW	Darkest shadow, values darker than this value will be set to this value
ICAP_XSCALING	Source Scaling value (1.0 = 100%) for x-axis
ICAP_YSCALING	Source Scaling value (1.0 = 100%) for y-axis

Image Type

ICAP_BITDEPTH	Pixel bit depth for current value of ICAP_PIXELTYPE
ICAP_BITDEPTHREDUCTION	Allows a choice of the reduction method for bit depth loss
ICAP_BITORDER	Specifies how the bytes in an image are filled by the Source
ICAP_CUSTHALFTONE	Square-cell halftone (dithering) matrix to be used
ICAP_HALFTONES	Source halftone patterns
ICAP_PIXELFLAVOR	Sense of the pixel whose numeric value is zero
ICAP_PIXELTYPE	The type of pixel data (B/W, gray, color, etc.)
ICAP_THRESHOLD	Specifies the dividing line between black and white values

Pages

ICAP_FRAMES	Size and location of frames on page
ICAP_MAXFRAMES	Maximum number of frames possible per page
ICAP_SUPPORTEDSIZES	Fixed frame sizes for typical page sizes

Resolution

ICAP_XNATIVERESOLUTION	Native optical resolution of device for x-axis
ICAP_XRESOLUTION	Current/Available optical resolutions for x-axis
ICAP_YNATIVERESOLUTION	Native optical resolution of device for y-axis
ICAP_YRESOLUTION	Current/Available optical resolutions for y-axis

Transfers

CAP_XFERCOUNT	Number of images the application is willing to accept this session
ICAP_COMPRESSION	Buffered Memory transfer compression schemes
ICAP_IMAGEFILEFORMAT	File formats for file transfers
ICAP_TILES	Tiled image data
ICAP_XFERMECH	Transfer mechanism - used to learn options and set-up for upcoming transfer
ICAP_UNDEFINEDIMAGESIZE	The application will accept undefined image size

User Interface

CAP_INDICATORS	Use the Source's progress indicator? (valid only when ShowUI==FALSE)
CAP_UICONTROLLABLE	Indicates that Source supports acquisitions with UI disabled

The Capability Listings

The following table lists descriptions of all TWAIN capabilities in alphabetical order. The format of each capability entry is:

NAME OF CAPABILITY

Description

Description of the capability

Application

(Optional) Information for the application

Source

(Optional) Information for the Source

Values

Type:	Data structure for capability
Default Value:	The value the Source must use as the current value when entering State 4 (following DG_CONTROL / DAT_IDENTITY / MSG_OPENDS). This is the value the Source resets the current value to when it receives a MSG_RESET operation. The Source reports its preferred default value when it receives a MSG_GETDEFAULT. The Source's preferred value may be different than the TWAIN default value.
Allowed Values:	Definition of the values allowed for this capability
Container for MSG_GET	Acceptable containers for use on MSG_GET operations
Container for MSG_SET	Acceptable containers for use on MSG_SET operations

Required By

If a Source or application is required to support the capability

Source Required Operations

Operations the Source is required to support.

See Also

Associated capabilities and data structures

CAP_AUTHOR

Description

The name or other identifying information about the Author of the image. It may include a copyright string.

Source

If not supported, the Source should return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_STR128
Default Value:	"\0"
Allowed Values:	any string
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_CAPTION
CAP_TIMEDATE

CAP_AUTOFEED

Description

If TRUE, the Source will automatically feed the next page from the document feeder after the number of frames negotiated for capture from each page are acquired. CAP_FEEDERENABLED must be TRUE to use this capability.

Application

Set the capability to TRUE to enable the Source's automatic feed process, or FALSE to disable it. After the completion of each transfer, check TW_PENDINGXFERS.Count to determine if the Source has more images to transfer. A -1 means there are more images to transfer but the exact number is not known.

CAP_FEEDERLOADED indicates whether the Source's feeder is loaded. (The automatic feed process continues whenever this capability is TRUE.)

Source

If CAP_FEEDERENABLED equals FALSE, return TWRC_FAILURE / TWCC_BADCAP (capability not supported in current settings).

If it is supported, return TWRC_SUCCESS and enable the device's automatic feed process: After all frames negotiated for capture from each page are acquired, put the current document in the output area and advance the next document from the input area to the feeder image acquisition area. If the feeder input area is empty, the automatic feeding process is suspended but should continue when the feeder is reloaded.

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

All Sources with Feeder Devices

Source Required Operations

MSG_GET/CURRENT/DEFAULT,
MSG_SET/RESET

See Also

CAP_CLEARPAGE
CAP_FEEDERENABLED
CAP_FEEDERLOADED
CAP_FEEDPAGE
CAP_REWINDPAGE

CAP_CAPTION

Description

A general note about the acquired image

Source

If not supported, the Source should return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_STR255
Default Value:	"\0"
Allowed Values:	any string
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_AUTHOR
CAP_TIMEDATE

CAP_CLEARPAGE

Description

If TRUE, the Source will eject the current page being acquired from and will leave the feeder acquire area empty.

If CAP_AUTOFEED is TRUE, a fresh page will be advanced.

CAP_FEEDERENABLED must equal TRUE to use this capability.

This capability must have been negotiated as an extended capability to be used in States 5 and 6.

Application

Do a MSG_SET on this capability to advance the document in the feeder acquire area to the output area and abort all transfers pending on this page.

This capability is used in States 5 and 6 by applications controlling the Source's feeder (usually without the Source user interface).

This capability can also be used while CAP_AUTOFEED equals TRUE to abort all remaining transfers on this page and continue with the next page.

Source

If CAP_FEEDERENABLED equals FALSE, return TWRC_FAILURE / TWCC_BADCAP (capability not supported in current settings).

If supported, advance the document in the feeder-acquire area to the output area and abort all pending transfers from this page.

The Source will perform this action once whenever the capability is MSG_SET to TRUE. The Source should then revert the current value to FALSE.

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_AUTOFEED
CAP_EXTENDEDCAPS
CAP_FEEDERENABLED
CAP_FEEDERLOADED
CAP_FEEDPAGE
CAP_REWINDPAGE

CAP_DEVICEONLINE

Description

If TRUE, the physical hardware (e.g., scanner, digital camera, image database, etc.) that represents the image source is attached, powered on, and communicating.

Application

This capability can be issued at any time to determine the availability of the image source hardware.

Source

The receipt of this capability request should trigger a test of the status of the physical link to the image source. The source should not assume that the link is still active since the last transaction, but should issue a transaction that actively tests this condition.

Values

Type:	TW_BOOL
Default Value:	None
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT

CAP_EXTENDEDCAPS

Description

Allows the Application and Source to negotiate capabilities to be used in States 5 and 6.

Application

MSG_GETCURRENT provides a list of all capabilities which the Source and application have agreed to negotiate in States 5 and 6.

MSG_GET provides a list of all capabilities the Source is willing to negotiate in States 5 and 6.

MSG_SET specifies which capabilities the application wants to negotiate in States 5 and 6.

Source

If not supported return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16
Default Value:	No Default
Allowed Values:	any xCAP_xxxx
Container for MSG_GET:	TW_ARRAY
Container for MSG_SET:	TW_ARRAY

Required By

None

Source Required Operations

None

See Also

CAP_SUPPORTEDCAPS

CAP_FEEDERENABLED

Description

If TRUE, Source must acquire data from the document feeder acquire area and other feeder capabilities can be used. If FALSE, Source must acquire data from the non-feeder acquire area and no other feeder capabilities can be used.

Application

The application should MSG_SET this capability to TRUE before attempting to use any other feeder capabilities. This sets the current acquire area to the feeder area (it may not be a different physical area on some Sources).

The application can MSG_SET this capability to FALSE to use the Source's non-feeder acquisition area and disallow the further use of feeder capabilities.

Source

This setting should reflect the current acquire area:

If TRUE - feeder acquire area should be used
If FALSE - use non-feeder acquire area

Usually, the feeder acquire area and non-feeder acquire area of the Source will be the same. For example, a flat bed scanner may feed a page onto the flatbed platen then scanning always takes place from the platen.

The counter example is a flatbed scanner that moves the scan bar over the platen when CAP_FEEDERENABLED is FALSE but moves the paper over the scan bar when it is TRUE.

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

All Sources with Feeder Devices

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

CAP_AUTOFEED
CAP_CLEARPAGE
CAP_FEEDERLOADED
CAP_FEEDPAGE
CAP_REWINDPAGE

Default Support Guidelines for Sources

- Flatbed scanner (without an optional ADF installed) - Default to FALSE. Do not allow setting to TRUE (return TWRC_FAILURE / TWCC_BADVALUE) but support the capability (never return TWRC_FAILURE / TWCC_BADCAP).
- A device that uses the same acquire area for feeder and non-feeder, and has a feeder installed - Default to TRUE and allow settings to TRUE or FALSE (meaning allow or don't allow other feeder capabilities).
- A device that operates differently when acquiring from the feeder and non-feeder areas (ex. physical pages sizes are different) - Default to preferred area and allow setting to either TRUE or FALSE.
- A sheet feed scanner or image database - Default to TRUE (meaning there is only one acquire area - the feeder area) and do not allow setting to FALSE (return TWRC_FAILURE / TWCC_BADVALUE).
- A handheld scanner would not support this capability (return TWRC_FAILURE / TWCC_BADCAP).

CAP_FEEDERLOADED

Description

Reflect whether there are documents loaded in the Source's feeder.

Application

Used by application to inquire whether there are documents loaded in the Source's feeder.

CAP_FEEDERENABLED must equal TRUE to use this capability.

Source

If CAP_FEEDERENABLED equals FALSE, return TWRC_FAILURE / TWCC_BADCAP (capability not supported in current settings).

If CAP_FEEDERENABLED equals TRUE, return the status of the feeder (documents loaded = TRUE, no documents loaded = FALSE).

The Source is responsible for reporting instructions to users on using the device. This includes instructing the user to place documents in the feeder when CAP_FEEDERLOADED equals FALSE and the application has requested a feed page (manually or automatically).

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

All Sources with Feeder Devices

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

CAP_AUTOFEED
CAP_CLEARPAGE
CAP_FEEDERENABLED
CAP_FEEDPAGE
CAP_REWINDPAGE

CAP_FEEDPAGE

Description

If TRUE, the Source will eject the current page and advance the next page in the document feeder into the feeder acquire area.

If CAP_AUTOFEED is TRUE, the same action just described will occur and CAP_AUTOFEED will remain active.

CAP_FEEDERENABLED must equal TRUE to use this capability.

This capability must have been negotiated as an extended capability to be used in States 5 and 6.

Application

Do a MSG_SET to TRUE on this capability to advance the next document in the feeder to the feeder acquire area.

This capability is used in States 5 and 6 by applications controlling the Source's feeder (usually without the Source's user interface).

This capability can also be used while CAP_AUTOFEED equals TRUE to abort all remaining transfers on this page and continue with the next page.

Source

If CAP_FEEDERENABLED equals FALSE, return TWRC_FAILURE / TWCC_BADCAP (capability not supported in current settings).

If supported, advance the document in the feeder-acquire area to the output area and abort all pending transfers from this page.

Advance the next page in the input area to the feeder acquire area. If there are no documents in the input area, return: TWRC_FAILURE / TWCC_BADVALUE.

The Source will perform this action once whenever the capability is MSG_SET to TRUE. The Source should then revert the current value to FALSE.

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_AUTOFEED
CAP_CLEARPAGE
CAP_EXTENDEDCAPS
CAP_FEEDERENABLED
CAP_FEEDERLOADED
CAP_REWINDPAGE

CAP_INDICATORS

Description

If TRUE, the Source will display a progress indicator during acquisition and transfer, regardless of whether the Source's user interface is active. If FALSE, the progress indicator will be suppressed if the Source's user interface is inactive.

The Source will continue to display device-specific instructions and error messages even with the Source user interface and progress indicators turned off.

Application

If the application plans to enable the Source with TW_USERINTERFACE.ShowUI = FALSE, it can also suppress the Source's progress indicator by using this capability.

Source

If supported, do not display progress indicator. If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_BOOL
Default Value:	TRUE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

CAP_REWINDPAGE

Description

If TRUE, the Source will return the current page to the input side of the document feeder and feed the last page from the output side of the feeder back into the acquisition area.

If CAP_AUTOFEED is TRUE, automatic feeding will continue after all negotiated frames from this page are acquired.

CAP_FEEDERENABLED must equal TRUE to use this capability.

This capability must have been negotiated as an extended capability to be used in States 5 and 6.

Application

This capability is used in States 5 and 6 by applications controlling the Source's feeder (usually without the Source's user interface).

If CAP_AUTOFEED is TRUE, the normal automatic feeding will continue after all frames of this page are acquired.

Source

If CAP_FEEDERENABLED equals FALSE, return TWRC_FAILURE / TWCC_BADCAP (capability not supported in current settings).

If there are no documents in the output area, return: TWRC_FAILURE / TWCC_BADVALUE.

The Source will perform this action once whenever the capability is MSG_SET to TRUE. The Source should then revert the current value to FALSE.

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

CAP_AUTOFEED
CAP_CLEARPAGE
CAP_EXTENDEDCAPS
CAP_FEEDERENABLED
CAP_FEEDERLOADED
CAP_FEEDPAGE

CAP_SUPPORTEDCAPS

Description

Returns a list of all the capabilities for which the Source will answer inquiries. Does not indicate which capabilities the Source will allow to be set by the application. Some capabilities can only be set if certain setup work has been done so the Source cannot globally answer which capabilities are "set-able."

Source

Values

Type:	TW_UINT16
Default Value:	No Default
Allowed Values:	Any "get-able" capability
Container for MSG_GET:	TW_ARRAY
Container for MSG_SET:	MSG_SET not allowed

Required By

All Sources.

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

CAP_EXTENDEDCAPS

CAP_TIMEDATE

Description

The date and time the image was acquired.

Stored in the form "YYYY/MM/DD HH:mm:SS.sss" where YYYY is the year, MM is the numerical month, DD is the numerical day, HH is the hour, mm is the minute, SS is the second, and sss is the millisecond.

Source

The time and date when the image was originally acquired (when Source entered State 7).

Be sure to leave the space between the ending of the date and beginning of the time fields. Pad the unused characters after the string with zeros.

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_STR32
Default Value:	No Default
Allowed Values:	Any date
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

None

Source Required Operations

None

See Also

CAP_AUTHOR
CAP_CAPTION

CAP_UICONTROLLABLE

Description

If TRUE, indicates that this Source supports acquisition with the UI disabled; i.e., TW_USERINTERFACE's ShowUI field can be set to FALSE. If FALSE, indicates that this Source can only support acquisition with the UI enabled.

Source

This capability is new to TWAIN 1.6. All Sources compliant with TWAIN 1.6 and above must support this capability. Sources that are not TWAIN 1.6-compliant may return TWRC_FAILURE / TWCC_BADCAP if they do not support this capability.

Application

This capability is new to TWAIN 1.6. A return value of TWRC_FAILURE / TWCC_BADCAP indicates that the Source in use is not TWAIN 1.6-compliant. Therefore, the Source may ignore TW_USERINTERFACE's ShowUI field when MSG_ENABLEDS is issued. See the description of DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS for more details.

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

All Sources.

See Also

CAP_INDICATORS
DG_CONTROL / DAT_USERINTERFACE / MSG_ENABLEDS

CAP_XFERCOUNT

Description

The application is willing to accept this number of images.

Application

Set this capability to the number of images you are willing to transfer per session. Common values:

1 Application wishes to transfer only one image this session

-1 Application is willing to transfer multiple images

Source

If the application limits the number of images it is willing to receive, the Source should not make more transfers available than the specified number.

Values

Type:	TW_INT16
Default Value:	-1
Allowed Values:	-1 to 2^{15}
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

All Sources and applications

Source Required Operations

MSG_GET/CURRENT/DEFAULT,
MSG_SET/RESET

See Also

TW_PENDINGXFERS.Count

ICAP_AUTOBRIGHT

Description

TRUE enables and FALSE disables the Source's Auto-brightness function (if any).

Source

If TRUE, apply auto-brightness function to acquired image before transfer.

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_BRIGHTNESS

ICAP_BITDEPTH

Description

Specifies the pixel bit depths for the current value of ICAP_PIXELTYPE. For example, when using ICAP_PIXELTYPE = TWPT_GRAY, this capability specifies whether this is 8-bit gray or 4-bit gray.

This depth applies to all the data channels (for instance, the R, G, and B channels will all have this same bit depth for RGB data).

Application

The application should loop through all the ICAP_PIXELTYPEs it is interested in and negotiate the ICAP_BITDEPTH(s) for each.

For all allowed settings of ICAP_PIXELTYPE

- Set ICAP_PIXELTYPE
- Set ICAP_BITDEPTH for the current ICAP_PIXELTYPE

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

If the bit depth in a MSG_SET is not supported for the current ICAP_PIXELTYPE setting, return TWRC_FAILURE / TWCC_BADVALUE.

Values

Type:	TW_UINT16
Default Value:	No Default
Allowed Values:	>=1
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

ICAP_PIXELTYPE

ICAP_BITDEPTHREDUCTION

Description

Specifies the Reduction Method the Source should use to reduce the bit depth of the data. Most commonly used with ICAP_PIXELTYPE = TWPT_BW to reduce gray data to black and white.

Application

Set the capability to the reduction method to be used in future acquisitions

Also select the Halftone or Threshold to be used.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWBR_THRESHOLD	0
	TWBR_HALFTONES	1
	TWBR_CUSTHALFTONE	2
	TWBR_DIFFUSION	3
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_CUSTHALFTONE
ICAP_HALFTONES
ICAP_PIXELTYPE
ICAP_THRESHOLD

ICAP_BITORDER

Description

Specifies how the bytes in an image are filled by the Source. TWBO_MSBFIRST indicates that the leftmost bit in the byte (usually bit 7) is the byte's Most Significant Bit.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16	
Default Value:	TWBO_MSBFIRST	
Allowed Values:	TWBO_LSBFIRST	0
	TWBO_MSBFIRST	1
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE	

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

ICAP_BITORDERCODES

Description

Used for CCITT data compression only. Indicates the bit order representation of the stored compressed codes.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16	
Default Value:	TWBO_LSBFIRST	
Allowed Values:	TWBO_LSBFIRST	0
	TWBO_MSBFIRST	1
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_COMPRESSION

ICAP_BRIGHTNESS

Description

The brightness values available within the Source.

Application

The application can use this capability to inquire, set or restrict the values for BRIGHTNESS used in the Source.

Source

Source should normalize the values into the range. Make sure that a '0' value is available as the Current Value when the Source starts up. If the Source's ± range is asymmetric about the '0' value, set range maxima to ±1000 and scale homogeneously from the '0' value in each direction. This will yield a positive range whose step size differs from the negative range's step size.

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	0
Allowed Values:	-1000 to +1000
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_AUTOBRIGHT
ICAP_CONTRAST

ICAP_CCITTKFACTOR

Description

Used for CCITT Group 3 2-dimensional compression. The 'K' factor indicates how often the new compression baseline should be re-established. A value of 2 or 4 is common in facsimile communication. A value of zero in this field will indicate an infinite K factor—the baseline is only calculated at the beginning of the transfer.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values:

Type:	TW_UINT16
Default Value:	4
Allowed Values:	0 to 2^{16}
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_COMPRESSION

ICAP_COMPRESSION

Description

Allows the application and Source to identify which compression schemes they have in common for Buffered Memory transfers.

Application

Applications must not assume that a Source can provide compressed, Buffered Memory transfers because many cannot.

The application should use MSG_SET on a TW_ONEVALUE container to specify the compression type for future transfers.

Source

The current value of this setting specifies the compression method to be used in future Buffered Memory transfers.

Values

Type:	TW_UINT16	
Default Value:	TWCP_NONE	
Allowed Values:	TWCP_NONE	0
	TWCP_PACKBITS	1
	TWCP_GROUP31D	2
	TWCP_GROUP31DEOL	3
	TWCP_GROUP32D	4
	TWCP_GROUP34	5
	TWCP_JPEG	6
	TWCP_LZW	7
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE	

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

DG_CONTROL / DAT_IMAGEMEMXFER / MSG_GET

ICAP_CONTRAST

Description

The contrast values available within the Source.

Application

The application can use this capability to inquire, set or restrict the values for CONTRAST used in the Source.

Source

Scale the values available internally into a homogeneous range between -1000 and 1000. Make sure that a '0' value is available as the Current Value when the Source starts up. If the Source's ± range is asymmetric about the '0' value, set range maxima to ±1000 and scale homogeneously from the '0' value in each direction. This will yield a positive range whose step size differs from the negative range's step size.

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	0
Allowed Values:	-1000 to +1000
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_BRIGHTNESS

ICAP_CUSTHALFTONE

Description

Specifies the square-cell halftone (dithering) matrix the Source should use to halftone the image.

Application

The application should also set ICAP_BITDEPTHREDUCTION to TWBR_CUSTHALFTONE to use this capability.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT8
Default Value:	No Default
Allowed Values:	any rectangular array
Container for MSG_GET:	TW_ARRAY
Container for MSG_SET:	TW_ARRAY

Required By

None

Source Required Operations

None

See Also

ICAP_BITDEPTHREDUCTION

ICAP_EXPOSURETIME

Description

Specifies the exposure time used to capture the image, in seconds.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	>0
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_FILTER

Description

Describes the color characteristic of the subtractive filter applied to the image data. Multiple filters may be applied to a single acquisition.

Source

If the Source only supports application of a single filter during an acquisition and multiple filters are specified by the application, set the current filter to the first one requested and return TWRC_CHECKSTATUS.

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWFT_RED	0
	TWFT_GREEN	1
	TWFT_BLUE	2
	TWFT_NONE	3
	TWFT_WHITE	4
	TWFT_CYAN	5
	TWFT_MAGENTA	6
	TWFT_YELLOW	7
	TWFT_BLACK	8
Container for MSG_GET:	TW_ARRAY TW_ONEVALUE	
Container for MSG_SET:	TW_ARRAY TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_FLASHUSED

Description

Specifies whether or not the image was acquired using a flash.

Application

Note that an image with flash may have a different color composition than an image without flash.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_FRAMES

Description

The list of frames the Source will acquire on each page.

Application

MSG_GET returns the size and location of all the frames the Source will acquire image data from when acquiring from each page.

MSG_GETCURRENT returns the size and location of the next frame to be acquired.

MSG_SET allows the application to specify the frames and their locations to be used to acquire from future pages.

This ICAP is most useful if the Source supports simultaneous acquisition from multiple frames. Use ICAP_MAXFRAMES to establish this ability.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FRAME
Default Value:	No Default
Allowed Values:	Device dependent
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_MAXFRAMES
ICAP_SUPPORTEDSIZES
TW_IMAGELAYOUT

ICAP_GAMMA

Description

Gamma correction value for the image data.

Application

Do not use with TW_CIECOLOR, TW_GRAYRESPONSE, or TW_RGBRESPONSE data.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	2.2
Allowed Values:	any value
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_HALFTONES

Description

A list of names of the halftone patterns available within the Source.

Application

The application may not rename any halftone pattern.

The application should also set ICAP_BITDEPTHREDUCTION to use this capability.

Values

Type:	TW_STR32
Default Value:	No Default
Allowed Values:	any halftone name
Container for MSG_GET:	TW_ARRAY(for backwards compatibility with 1.0 only) TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	TW_ARRAY(for backwards compatibility with 1.0 only) TW_ENUMERATION TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_CUSTHALFTONE
ICAP_BITDEPTHREDUCTION
ICAP_THRESHOLD

ICAP_HIGHLIGHT

Description

Specifies which value in an image should be interpreted as the lightest "highlight." All values "lighter" than this value will be clipped to this value. Whether lighter values are smaller or larger can be determined by examining the current value of ICAP_PIXELFLAVOR.

Source

If more or less than 8 bits are used to describe the image, the actual data values should be normalized to fit within the 0-255 range. The normalization need not result in a homogeneous distribution if the original distribution was not homogeneous.

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	255
Allowed Values:	0 to 255
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_IMAGEFILEFORMAT

Description

Informs the application which file format(s) the Source can generate (MSG_GET). Tells the Source which file format(s) the application can handle (MSG_SET).

Application

Use this ICAP to determine which formats are available for file transfers, but use the DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET operation to specify the format to be used for a particular acquisition.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16
Default Value:	TWFF_BMP (Windows) TWFF_PICT (Macintosh)
Allowed Values:	TWFF_TIFF 0 TWFF_PICT 1 TWFF_BMP 2 TWFF_XBM 3 TWFF_JFIF 4
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

DG_CONTROL / DAT_SETUPFILEXFER / MSG_SET

DG_IMAGE / DAT_IMAGEFILEXFER / MSG_GET

ICAP_JPEGPIXELTYPE

Description

Allows the application and Source to agree upon a common set of color descriptors that are made available by the Source. This ICAP is only useful for JPEG-compressed Buffered Memory image transfers.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWPT_BW	0
	TWPT_GRAY	1
	TWPT_RGB	2
	TWPT_PALETTE	3
	TWPT_CMY	4
	TWPT_CMYK	5
	TWPT_YUV	6
	TWPT_YUVK	7
	TWPT_CIEXYZ	8
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_COMPRESSION

ICAP_LAMPSTATE

Description

TRUE means the lamp is currently, or should be set to, ON. Sources may not support MSG_SET operations.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_LIGHTPATH

Description

Describes whether the image was captured transmissively or reflectively.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWLP_REFLECTIVE	0
	TWLP_TRANSMISSIVE	1
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_LIGHTSOURCE

Description

Describes the general color characteristic of the light source used to acquire the image.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWLS_RED	0
	TWLS_GREEN	1
	TWLS_BLUE	2
	TWLS_NONE	3
	TWLS_WHITE	4
	TWLS_UV	5
	TWLS_IR	6
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_MAXFRAMES

Description

The maximum number of frames the Source can provide or the application can accept per page.

This is a bounding capability only. It does not establish current or future behavior.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16
Default Value:	No Default
Allowed Values:	1 to 2^{16}
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_FRAMES
TW_IMAGELAYOUT

ICAP_ORIENTATION

Description

Defines which edge of the "paper" the image's "top" are aligned with. The upper-left of the image is defined as the location where both the primary and secondary scans originate. (The X axis is the primary scan direction and the Y axis is the secondary scan direction.) For a flatbed scanner, the light bar moves in the secondary scan direction. For a hand-held scanner, the scanner is drug in the secondary scan direction. For a digital camera, the secondary direction is the vertical axis when the viewed image is considered upright.

Application

If one pivots the image about its center, then orienting the image in TWOR_LANDSCAPE has the effect of rotating the original image 90 degrees to the "left." TWOR_PORTRAIT mode does not rotate the image. The image may be oriented along any of the four axes located 90 degrees from the unrotated image. Note that:

TWOR_ROT0 == TWOR_PORTRAIT and TWOR_ROT270 == TWOR_LANDSCAPE.

Source

The Source is responsible for rotating the image if it allows this capacity to be set.

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16	
Default Value:	TWOR_PORTRAIT	
Allowed Values:	TWOR_ROT0	0
	TWOR_ROT90	1
	TWOR_ROT180	2
	TWOR_ROT270	3
	TWOR_PORTRAIT	(equals TWOR_ROT0)
	TWOR_LANDSCAPE	(equals TWOR_ROT270)
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE	

Required By

None

Source Required Operations

None

See Also

ICAP_PHYSICALHEIGHT

Description

The maximum physical height (Y-axis) the Source can acquire (measured in units of ICAP_UNITS).

Source

For a flatbed scanner, the scannable height of the platen. For a hand-held scanner, the maximum length of a scan.

For dimensionless devices, such as digital cameras, this ICAP is meaningless for all values of ICAP_UNITS other than TWUN_PIXELS. If the device is dimensionless, the Source should return a value of zero if ICAP_UNITS does not equal TWUN_PIXELS. This tells the application to inquire with TWUN_PIXELS.

Note: The physical acquired area may be different depending on the setting of ICAP_FEEDERENABLED (if the Source has separate feeder and non-feeder acquire areas).

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	0 to 65535 in ICAP_UNITS
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

ICAP_FEEDERENABLED
ICAP_UNITS

ICAP_PHYSICALWIDTH

Description

The maximum physical width (X-axis) the Source can acquire (measured in units of ICAP_UNITS).

Source

For a flatbed scanner, the scannable width of the platen. For a hand-held scanner, the maximum width of a scan.

For dimensionless devices, such as digital cameras, this ICAP is meaningless for all values of ICAP_UNITS other than TWUN_PIXELS. If the device is dimensionless, the Source should return a value of zero if ICAP_UNITS does not equal TWUN_PIXELS. This tells the application to inquire with TWUN_PIXELS. The Source should then reply with its X-axis pixel count.

Note: The physical acquired area may be different depending on the setting of ICAP_FEEDERENABLED (if the Source has separate feeder and non-feeder acquire areas).

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	0 to 65535 in ICAP_UNITS
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

ICAP_FEEDERENABLED
ICAP_UNITS

ICAP_PIXELFLAVOR

Description

Sense of the pixel whose numeric value is zero (minimum data value).

For example, consider a black and white image:

If ICAP_PIXELTYPE is TWPT_BW then
  If ICAP_PIXELFLAVOR is TWPF_CHOCOLATE
    then Black = 0
  Else if ICAP_PIXELFLAVOR is TWPF_VANILLA
    then White = 0

Application

Sources may prefer a different value depending on ICAP_PIXELTYPE. Set ICAP_PIXELTYPE and do a MSG_GETDEFAULT to determine the Source's preferences.

Source

TWPF_CHOCOLATE means this pixel represents the darkest data value that can be generated by the device (the darkest available optical value may measure greater than 0).

TWPF_VANILLA means this pixel represents the lightest data value that can be generated by the device (the lightest available optical value may measure greater than 0).

Values

Type:	TW_UINT16	
Default Value:	TWPF_CHOCOLATE	
Allowed Values:	TWPF_CHOCOLATE	0
	TWPF_VANILLA	1
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE	

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT,
MSG_SET/RESET

See Also

ICAP_PIXELTYPE

ICAP_PIXELFLAVORCODES

Description

Used only for CCITT data compression. Specifies whether the compressed codes' pixel "sense" will be inverted from the current value of ICAP_PIXELFLAVOR prior to transfer.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16	
Default Value:	TWPF_CHOCOLATE	
Allowed Values:	TWPF_CHOCOLATE	0
	TWPF_VANILLA	1
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE	
Container for MSG_SET:	TW_ONEVALUE	

Required By

None

Source Required Operations:

None

See Also

ICAP_COMPRESSION

ICAP_PIXELTYPE

Description

The type of pixel data that a Source is capable of acquiring (ex. black and white, gray, RGB, etc.).

Application

- MSG_GET returns a list of all pixel types available from the Source.
- MSG_SET on a TW_ENUMERATION structure requests that the Source restrict the available pixel types to the enumerated list.
- MSG_SET on a TW_ONEVALUE container specifies the only pixel type the application can accept.

If the application plans to transfer data through any mechanism other than Native and cannot handle all possible ICAP_PIXELTYPEs, it **must** support negotiation of this ICAP.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWPT_BW	0
	TWPT_GRAY	1
	TWPT_RGB	2
	TWPT_PALETTE	3
	TWPT_CMY	4
	TWPT_CMYK	5
	TWPT_YUV	6
	TWPT_YUVK	7
	TWPT_CIEXYZ	8
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE	

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT,
MSG_SET/RESET

See Also

ICAP_BITDEPTH
ICAP_BITDEPTHREDUCTION

ICAP_PLANARCHUNKY

Description

Allows the application and Source to identify which color data formats are available. There are two options, "planar" and "chunky."

For example, planar RGB data is transferred with the entire red plane of data first, followed by the entire green plane, followed by the entire blue plane (typical for three-pass scanners). "Chunky" mode repetitively interlaces a pixel from each plane until all the data is transferred (R-G-B-R-G-B…) (typical for one-pass scanners).

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16
Default Value:	No Default
Allowed Values:	TWPC_CHUNKY 0 TWPC_PLANAR 1
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT

See Also

TW_IMAGEINFO.Planar

ICAP_ROTATION

Description

How the Source can/should rotate the image data prior to transfer. This doesn't use ICAP_UNITS. It is always measured in degrees. Any applied value is additive with any rotation specified in ICAP_ORIENTATION.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	0
Allowed Values:	+/- 360 degrees
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_SHADOW

Description

Specifies which value in an image should be interpreted as the darkest "shadow." All values "darker" than this value will be clipped to this value.

Application

Whether darker values are smaller or larger can be determined by examining the current value of ICAP_PIXELFLAVOR.

Source

If more or less than 8 bits are used to describe the image, the actual data values should be normalized to fit within the 0-255 range. The normalization need not result in a homogeneous distribution if the original distribution was not homogeneous.

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	0
Allowed Values:	0 to 255
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_PIXELFLAVOR

ICAP_SUPPORTEDSIZES

Description

For devices that support fixed frame sizes. Defined sizes match typical page sizes. This specifies the size(s) the Source can/should use to acquire image data.

Source

The frame size selected by using this capability should be reflected in the TW_IMAGELAYOUT structure information.

If the Source cannot acquire the exact frame size specified by the application, it should provide the closest possible size (preferably acquiring an image that is larger than the requested frame in both axes).

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16	
Default Value:	No Default	
Allowed Values:	TWSS_NONE	0
	TWSS_A4LETTER	1
	TWSS_B5LETTER	2
	TWSS_USLETTER	3
	TWSS_USLEGAL	4
	TWSS_A5	5
	TWSS_B4	6
	TWSS_B6	7
	TWSS_B	8
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE	

Required By

All Image Sources that support fixed frame sizes

Source Required Operations

MSG_GET/CURRENT/DEFAULT,
MSG_SET/RESET

See Also

ICAP_FRAMES
TW_IMAGEINFO
TW_IMAGELAYOUT

ICAP_THRESHOLD

Description

Specifies the dividing line between black and white. This is the value the Source will use to threshold, if needed, when ICAP_PIXELTYPE = TWPT_BW.

The value is normalized so there are no units of measure associated with this ICAP.

Application

Application will typically set ICAP_BITDEPTHREDUCTION to TWBR_THRESHOLD to use this capability.

Source

Source should fit available values linearly into the defined range such that the lowest available value equals 0 and the highest equals 255.

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	128
Allowed Values:	0 to 255
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_BITDEPTHREDUCTION

ICAP_TILES

Description

This is used with Buffered Memory transfers. If TRUE, Source can provide application with tiled image data.

Application

If set to TRUE, the application expects the Source to supply tiled data for the upcoming transfer(s). This persists until the application sets it to FALSE. If the application sets it to FALSE, Source will supply strip data.

Source

If Source can supply tiled data and application does not set this ICAP, Source may or may not supply tiled data at its discretion.

In State 6, ICAP_TILES should reflect whether tiles or strips will be used in the upcoming transfer.

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_BOOL
Default Value:	No Default
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

TW_IMAGEMEMXFER

ICAP_TIMEFILL

Description

Used only with CCITT data compression. Specifies the minimum number of words of compressed codes (compressed data) to be transmitted per line.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16
Default Value:	1
Allowed Values:	1 to 2^{16}
Container for MSG_GET:	TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

ICAP_COMPRESSION

ICAP_UNDEFINEDIMAGESIZE

Description

If TRUE the Source will issue a MSG_XFERRDY before starting the scan.

Note: The Source may need to scan the image before initiating the transfer. This is the case if the scanned image is rotated or merged with another scanned image.

Application

Used by the application to notify the source that the application accepts -1 as the image width or -length in the TW_IMAGEINFO structure.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_BOOL
Default Value:	FALSE
Allowed Values:	TRUE or FALSE
Container for MSG_GET:	TW_ONEVALUE
Container for MSG_SET:	TW_ONEVALUE

Required By

None

Source Required Operations

None

See Also

TW_IMAGEINFO

ICAP_UNITS

Description

Unless a quantity is dimensionless or uses a specified unit of measure, ICAP_UNITS determines the unit of measure for all quantities.

Application

Applications should be able to handle TWUN_PIXELS if they want to support data transfers from "dimensionless" devices such as digital cameras.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_UINT16	
Default Value:	TWUN_INCHES	
Allowed Values:	TWUN_INCHES	0
	TWUN_CENTIMETERS	1
	TWUN_PICAS	2
	TWUN_POINTS	3
	TWUN_TWIPS	4
	TWUN_PIXELS	5
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE	

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT,
MSG_SET/RESET

See Also

ICAP_XFERMECH

Description

Allows the application and Source to identify which transfer mechanisms they have in common.

Application

The current setting of ICAP_XFERMECH must match the constant used by the application to specify the transfer mechanism when starting the transfer using the triplet: DG_IMAGE / DAT_IMAGExxxxXFER / MSG_GET.

Values

Type:	TW_UINT16	
Default Value:	TWSX_NATIVE	
Allowed Values:	TWSX_NATIVE	0
	TWSX_FILE	1
	TWSX_MEMORY	2
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE	
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE	

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT,
MSG_SET/RESET

See Also

DG_IMAGE / DAT_IMAGExxxXFER / MSG_GET

ICAP_XNATIVERESOLUTION

Description

The native optical resolution along the X-axis of the device being controlled by the Source. Most devices will respond with a single value (TW_ONEVALUE).

This is NOT a list of all resolutions that can be generated by the device. Rather, this is the resolution of the device's optics. Measured in units of pixels per unit as defined by ICAP_UNITS (pixels per TWUN_PIXELS yields dimensionless data).

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	>0
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

None

Source Required Operations

None

See Also

ICAP_UNITS
ICAP_XRESOLUTION
ICAP_YNATIVERESOLUTION

ICAP_XRESOLUTION

Description

All the X-axis resolutions the Source can provide.

Measured in units of pixels per unit as defined by ICAP_UNITS (pixels per TWUN_PIXELS yields dimensionless data).

Application

Setting this value will restrict the various resolutions that will be available to the user during acquisition.

Applications will want to ensure that the values set for this ICAP match those set for ICAP_YRESOLUTION.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	>0
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT,
MSG_SET/RESET

See Also

ICAP_UNITS
ICAP_XNATIVERESOLUTION
ICAP_YRESOLUTION

ICAP_XSCALING

Description

All the X-axis scaling values available. A value of '1.0' is equivalent to 100% scaling. Do not use values less than or equal to zero.

Application

Applications will want to ensure that the values set for this ICAP match those set for ICAP_YSCALING. There are no units inherent with this data as it is normalized to 1.0 being "unscaled."

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP

Values

Type:	TW_FIX32
Default Value:	1.0
Allowed Values:	> 0
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_YSCALING

ICAP_YNATIVERESOLUTION

Description

The native optical resolution along the Y-axis of the device being controlled by the Source.

Measured in units of pixels per unit as defined by ICAP_UNITS (pixels per TWUN_PIXELS yields dimensionless data).

Application

Most devices will respond with a single value (TW_ONEVALUE). This is NOT a list of all resolutions that can be generated by the device. Rather, this is the resolution of the device's optics

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	> 0
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE
Container for MSG_SET:	MSG_SET not allowed

Required By

None

Source Required Operations

None

See Also

ICAP_UNITS
ICAP_XNATIVERESOLUTION
ICAP_YRESOLUTION

ICAP_YRESOLUTION

Description

All the Y-axis resolutions the Source can provide.

Measured in units of pixels per unit as defined by ICAP_UNITS (pixels per TWUN_PIXELS yields dimensionless data).

Application

Setting this value will restrict the various resolutions that will be available to the user during acquisition.

Applications will want to ensure that the values set for this ICAP match those set for ICAP_XRESOLUTION.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	No Default
Allowed Values:	> 0
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

All Image Sources

Source Required Operations

MSG_GET/CURRENT/DEFAULT,
MSG_SET/RESET

See Also

ICAP_UNITS
ICAP_XRESOLUTION
ICAP_YNATIVERESOLUTION

ICAP_YSCALING

Description

All the Y-axis scaling values available. A value of '1.0' is equivalent to 100% scaling. Do not use values less than or equal to zero.

There are no units inherent with this data as it is normalized to 1.0 being "unscaled."

Application

Applications will want to ensure that the values set for this ICAP match those set for ICAP_XSCALING.

Source

If not supported, return TWRC_FAILURE / TWCC_BADCAP.

Values

Type:	TW_FIX32
Default Value:	1.0
Allowed Values:	> 0
Container for MSG_GET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE
Container for MSG_SET:	TW_ENUMERATION TW_ONEVALUE TW_RANGE

Required By

None

Source Required Operations

None

See Also

ICAP_XSCALING

10

Return Codes and Condition Codes

Chapter Contents

An Overview of Return Codes and Condition Codes

The TWAIN protocol defines no dynamic messaging system through which the application might determine, in real-time, what is happening in either the Source Manager or a Source. Neither does the protocol implement the native messaging systems built into the operating environments that TWAIN is defined to operate under (MS Windows and Macintosh). This decision was made due to issues regarding platform specificity and higher-than-desired implementation costs.

Instead, for each call the application makes to DSM_Entry( ), whether aimed at the Source Manager or a Source, the Source Manager returns an appropriate Return Code (TWRC_xxx). The Return Code may have originated from the Source if that is where the original operation was destined.

To get more specific status information, the application can use the DG_CONTROL / DAT_STATUS / MSG_GET operation to inquire the complimentary Condition Code (TWCC_xxx) from the Source Manager or Source (whichever one originated the Return Code).

The application should always check the Return Code. If the Return Code is TWRC_FAILURE, it should also check the Condition Code. This is especially important during capability negotiation.

There are very few, if any, catastrophic error conditions for the application to worry about. Usually, the application will only have to "recover" from low memory errors caused from allocations in the Source. Most error conditions are handled by the Source Manager or, most typically, by the Source (often involving interaction with the user). If the Source fails in a way that is unrecoverable, it will ask to have its user interface disabled by sending the MSG_CLOSEDSREQ to the application's event loop.

Currently Defined Return Codes

The following are the currently defined return codes:

TWRC_CANCEL	Abort transfer or the Cancel button was pressed.
TWRC_CHECKSTATUS	Partially successful operation; request further information.
TWRC_DSEVENT	Event (or Windows message) belongs to this Source.
TWRC_ENDOFLIST	No more Sources found after MSG_GETNEXT.
TWRC_FAILURE	Operation failed - get the Condition Code for more information.
TWRC_NOTDSEVENT	Event (or Windows message) does not belong to this Source.
TWRC_SUCCESS	Operation was successful.
TWRC_XFERDONE	All data has been transferred.

Currently Defined Condition Codes

The following are the currently defined condition codes:

TWCC_BADCAP*	Capability not supported by Source or operation (get, set) is not supported on capability, or capability had dependencies on other capabilities and cannot be operated upon at this time (Obsolete, see TWCC_CAPUNSUPPORTED, TWCC_BAPBADOPERATION, and TWCC_CAPSEQERROR).
TWCC_BADDEST	Unknown destination in DSM_Entry.
TWCC_BADPROTOCOL	Unrecognized operation triplet.
TWCC_BADVALUE	Data parameter out of supported range.
TWCC_BUMMER	General failure. Unload Source immediately.
TWCC_CAPUNSUPPORTED*	Capability not supported by Source.
TWCC_CAPBADOPERATION*	Operation (i.e., Get or Set) not supported on capability.
TWCC_CAPSEQERROR*	Capability has dependencies on other capabilities and cannot be operated upon at this time.
TWCC_LOWMEMORY	Not enough memory to complete operation.
TWCC_MAXCONNECTIONS	Source is connected to maximum supported number of applications.
TWCC_NODS	Source Manager unable to find the specified Source.
TWCC_OPERATIONERROR	Source or Source Manager reported an error to the user and handled the error; no application action required.

TWCC_SEQERROR	Illegal operation for current Source Manager or Source state.
TWCC_SUCCESS	Operation worked.

* TWCC_BADCAP has been replaced with three new condition codes that more clearly specify the reason for a capability operation failure. For backwards compatibility applications should also accept TWCC_BADCAP and treat it as a general capability operation failure. No 1.6 Image Data Sources should return this condition code, but use the new ones instead.

Custom Return and Condition Codes

Although probably not necessary or desirable, it is possible to create custom Return Codes and Condition Codes. Refer to the TWAIN.H file for the value of TWRC_CUSTOMBASE for custom Return Codes and TWCC_CUSTOMBASE for custom Condition Codes. All custom values must be numerically greater than these base values. Remember that the consumer of these custom values will look in your TW_IDENTITY.ProductName field to clarify what the identifier's value means. There is no other protection against overlapping custom definitions.

11

TWAIN Technical Support

Chapter Contents

E-Mail Support

Developers who are connected to AppleLink, CompuServe, the WWW or Internet have access to TWAIN support groups. The support groups can answer your TWAIN development or marketing questions. There are two support groups: the TWAIN Working Group and the TWAIN Developers distribution.

- The TWAIN Working Group is read by Technical, Marketing and Support representatives from the Working Group companies.
- The TWAIN Developers distribution includes TWAIN developers who want to keep up on TWAIN or offer advice to other developers. This distribution includes the TWAIN Working Group. This distribution is the best place to get support because both the Working Group and other developers can respond.

TWAIN developers are encouraged to participate on the TWAIN Developer distribution list . All developers responding to questions posted to this distribution should Cc the distribution. The TWAIN Working Group also uses this distribution as a means to communicate with developers. For example, we use the distribution when posting the latest news about TWAIN, asking questions we may have about implementations, and requesting review of any Technical Notes which are under development. Technical Notes provide the mechanism for distributing updated information and corrections to errors that may occur in this document.

For more information on e-mail addresses, please look at the TWAIN.FAQ file provided with this toolkit.

CompuServe

Developers connected to CompuServe can also get on-line information and updates including: the Implementer's Matrix, TWAIN Source Managers, TWAIN Sample Code, and the latest version information.

For more information on accessing CompuServe, please look at the TWAIN.FAQ file provide with this toolkit.

Worldwide Web

Developers connected to the WWW can also get on-line information and updates. There is an on-line version of the Implementor's matrix with connections to those implementers with WWW pages. In addition, this manual is available as a readable file.

The WWW address is: **http://www.twain.org/**

Information by Fax

From Hewlett-Packard

A short informational white paper on TWAIN and a TWAIN Developer's Toolkit Order Form are available using Hewlett-Packard's fax back system, HP FIRST. To receive these documents call from a touch tone phone or fax machine and information will be faxed to you.

Phone Numbers:

Inside the US or Canada	800 333-1917
Other locations	08 344-4809

The Document Numbers are:

3130	TWAIN Toolkit Order Form
3129	TWAIN White Paper (includes toolkit order form)

From Logitech

Logitech also offers a Faxback service for information about TWAIN.

Phone Number:

Inside the US or Canada	800 245-0000

The Document Numbers are:

4713	One-page TWAIN fact sheet
4714	TWAIN Implementer's Matrix
4715	TWAIN Implementer's Matrix Application
4718	TWAIN Toolkit Order Form
4719	TWAIN White Paper

Ordering Information

Inside the US and Canada, the TWAIN toolkit is available by calling: 800 722-0379. There is a nominal charge for the toolkit to cover manufacturing and distribution costs.

Outside the US and Canada, the TWAIN toolkit is available using the order form available by fax (see above).

A

Adobe Color Description Specification

The Working Group has proposed that the color descriptors defined in the following document from Adobe Systems, Inc. be the primary color descriptor set for TWAIN. This document does not specify a particular color space, though the descriptor's map into the generalized CIE 1931 (XYZ)-space. This generalized space has been less controversial than most other offshoots of this space or other descriptor types. We believe that this solution is adequate for the current state of the imaging market relative to TWAIN. Your comments about this set of descriptor's are welcome.

Note: This document was captured with OCR technology. There maybe some inverted errors from the process.

Adobe Color Space Descriptors

Adobe Systems, Incorporated

This revision was written Wednesday, October 9, 1991.

This paper proposes the basics of a color management system based on *color space descriptors.* These descriptors define the mapping from an imaging color space to the CIE 1931 (XYZ)-space. When used properly in conjunction with PostScript@ Level 2, they provide a method for specifying color in a relatively device independent manner.

Background

The quest for device independent color is hampered by the pressure to work in the native color space of particular devices. This pressure is particularly acute with respect to interactive display devices-i.e., computer monitors-where responsiveness is paramount. Adobe's proposed solution, based on the notion of a color space descriptor as defined in PostScript@ Level 2, allows individual devices to work in whatever color space they choose (from a range of color spaces which encompasses virtually all of the traditional spaces) provided that they include a description of this color space whenever they transmit the color data (images and graphic elements) to other devices.

For example, a scanner or a scanner driver will include a color space descriptor defining its characteristics along with the image data. An application can then use this descriptor and the

descriptor for the computer's monitor to convert the image data into the monitor's color space. When the image is transmitted to a printer, the computer can include either the scanner's descriptor and the original data or the monitor's descriptor and the converted data. The printer will then convert the image data to its own color space before printing. Within the bounds of color gamut and other device limitations, and to the extent that the color space descriptors accurately describe their respective devices, the result will be significantly improved color matching.

We do not address issues of color gamut or descriptor accuracy in this proposal. We merely focus on providing a way to accurately describe colors. This, however, is a significant step forward over current conditions.

Color space descriptors

A PostScript CIE-based color space is defined using a two-stage non-linear transformation from an arbitrary three dimensional space into CIE 1931 (XYZ)-space. (See pages 186-191 of the second edition of the *Postscript Language Reference Manual* or see Figure 1). As the manual demonstrates, these descriptors can describe a wide variety of color spaces.

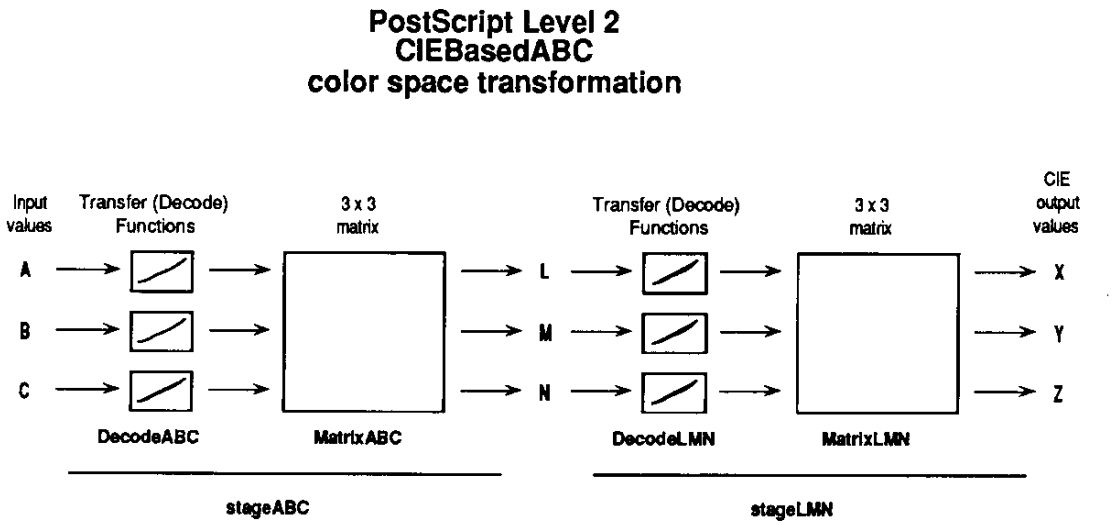


Figure 1

Colors described in one of these spaces have three components: A, B, and C. In existing operating system drawing environments such as QuickDraw and Windows GDI, these can be identified with the red, green, and blue components provided in the environment's color model. This is a reasonable assumption since the most frequent use for these descriptors will be to describe images presented in a calibrated RGB space (the monitor).

The descriptor could, however, describe Lab or HSL or a number of other spaces having no simple relationship to RGB. The colors specified in the operating system graphics environment would then have little or no relationship to the actual RGB values and would produce garbage if treated as RGB data. We do not see a need to protect people from doing so, however, since

drawing in some color space other than calibrated RGB will be an explicit choice made by the programmer or the data supplier.[1]

We do, however, see a need to provide some simple information about color space compatibility along with each descriptor. Hence, we propose including a 4 character (Macintosh OSType) value as part of each color space descriptor to indicate the general class of color space the described space belongs to. For example, a value of `RGB I will indicate that the space is a calibrated RGB space. Other values will be defined as the need arises. An application or printer driver can examine this type tag and determine whether it knows how to deal with such a color space. If it does not, it should probably generate an appropriate warning to the user.

Each stage of the transformation consists of channel-specific, monotonic PostScript functions (referred to as 'decode' or 'transfer' functions) being applied to each of the three channels of color information followed by multiplication of the resulting vector by a 3x3 matrix to produce three mixed channels.

Mathematically, we have:

$$(a, b, c) \rightarrow (d_A(a), d_B(b), d_C(c)) \times M$$

Hence, to define a stage in the transition, we need to specify the three functions and the 9 values in the matrix. Representing the values in the matrix is straightforward so we shift our attention to the problem of representing the functions.

PostScript@ Level 2 supports arbitrary functions (the only condition being that they have to be monotonic). One solution to the problem of representing functions then would be to include strings containing PostScript code in our descriptors. This, however, is very inconvenient since we are interested in allowing system software and/or applications to perform the appropriate conversions and we do not want to force them to parse and interpret PostScript. Furthermore, to speed conversions, we prefer that the functions be reasonably easy to invert, and arbitrary PostScript code certainly does not meet this need.

Hence, we choose to back away from the full power of Level 2 color space descriptors by limiting the decode functions we use. This proposal chooses two function classes which seem reasonably powerful and cover most of the cases which are likely to arise.

The first option is an extended form of the gamma function (x^{γ}). This function is frequently used in describing monitor and printing characteristics and is very simple to compute. We extend this function by adding an initial linear segment.

We specify this function by giving three points:

- The start point
- The point at which we break from a linear function to the gamma function the end point
- The gamma value.

If we denote these as start, break, end, and γ, we have the function:

---

[1] A scanner manufacturer might choose to produce data In a color space other than calibrated RGB, but, In the short term, this would be asking for trouble since currently most programs are not equipped to deal with non-RGB data. We see no reason, however, to arbitrarily limit people's power and expect that support for alternate color models will spread.

$d_\gamma(x \varepsilon [start.in .. break.in]) =$
(break.out - start.out)(x - start.in)/(break.in - start.in) + start.out

$d\gamma(x \varepsilon [break.in .. end\text{-}in]) = a[(x - start.in) / (end.in - start.in)]\gamma + c$
where a = (end.out - break.out) / (1 - [(break.in - start.in) / (end.in - start.in)]γ)
and c = end.out - a

We require that:

start.in < break-in < end.in

and that either

start.out < break-out < end.out

or

start.out > break.out > end.out.

The second option is table lookup with linear interpolation. We specify a starting and ending value on the input side, a sample count n ≥ 1, and a monotonic sequence of n+1 sample points ($S_{i\varepsilon\ 0..n}$). These sample points are to be equally spaced across the interval from start.in to end.in with S_o and S_o being the starting and ending values respectively. The resulting function is defined as follows:

step = (end.in - start.in) / n

ds (x ε [start.in .. end.in)) = [(x - start.in - k * step) / step] * (sk+1 - Sk + Sk
where k = ⌊(x - start.in) / step⌋

ds (end.in) = Sn

Here is the C definition for a decode function:

```
typedef Fixed int32 ;[2]
typedef
      struct TDecodeFunction
              {  Fixed         startIn, breakIn, endIn;
                 Fixed         startOut, breakout, endOut;
                 Fixed         gamma;
                 int32         sampleCount; // = 0 if we should use gamma
              }  TDecodeFunction;
```

The samples themselves will be stored elsewhere when we build a color descriptor so that we do not have to worry about variable length elements within the color space descriptor. Simply assume for now that if samplecount > 0, then somewhere we have an array of Fixeds containing the samples. We assume table lookup unless sampleCount is zero. In any event, startIn, endIn, startout, and endout must always be valid.

---

[2] We use lnt16 and int32 to represent 16-bit and 32-bit signed integers. Our structures are based on 16.16 fixed point numbers. We choose fixed point numbers over real numbers because fixed point numbers are easier to standardize-particularly across platforms.

The channel mixing is done using a R3 matrix. This matrix together with the three decode functions completely defines one stage in the color space transformation. Here is the C definition for a single stage.

```
typedef
struct TTransformationStage
      { TDecodeFunction   decode[3];
              Fixed            mix[3](3];  // rows x columns
      } TTransformationStage;
```

A color space transformation consists of two of these stages. PostScript refers to these stages as ABC and LMN. We keep this naming convention in the structure presented below.

PostScript color space descriptors also require a white point color and allow for a black point color. We require that both be specified since this simplifies working with color space descriptors. Here is the structure for specifying these points in CIE 1931 (XYZ)-space:

```
typedef
      struct TCIEPoint ( Fixed x,y,z; } TCIEPoint;
```

All that remains to specify a color space is to specify storage for the samples for sampled decode functions. We handle this by making color space descriptors variable length structures and including the samples after all of the other data. We put the sample tables end-to-end starting with the samples for the first ABC-stage decode function and ending with the samples for the third LMN-stage decode function.

As a final generalization to the structure, we add two flags and one field to the structure.

The first of these flags addresses the vexing issue of byte-order in storage of multibyte values. It is essentially a high-endian/low-endian flag. We use a 16-bit integer to hold this value and specify that zero means high byte first and any non-zero value means low-byte first.

The second flag allows us to use the color space class value in color space descriptors to label non-CIE 1931 (XYZ)-space based color spaces. For example, a scanner or separation software may produce device dependent CMYK values. By setting the device dependent flag to a non-zero value, we specify that only the color space class contains useful information and that the rest of the descriptor structure is to be ignored.

The field is a version number indicating the version of the color space descriptor specification implemented in this record. We do not anticipate future revisions at this time, but it is easier to add the field now than later. The current revision number is zero. This number is to be stored in a 32-bit field (for alignment purposes) in the byte-order indicated by the low-endian flag.

Here then is the structure of a color space descriptor. Note that the size of the samples field is arbitrary:

```
typedef char OSType [4];
#define kCalibratedRGB 'RGB '
#define kDeviceCMYK 'cmyk'
#define kCurrentDescriptorVersion 0

typedef
      struct TCIEBasedColorSpace
      {  OSType                 colorSpaceClass;
         int16                  lowEndian;
         int16                  deviceDependent;
         int32                  versionNumber;
         TTransformationStage   stageABC;
         TTransformationStage   stageLMN;
         TCIEPoint              whitePoint;
         TCIEPoint              blackPoint;
         Fixed                  samples[0];
      } TCIEBasedColorSpace;
```

Conclusions

Adobe is proposing a standard method for describing color spaces and thereby providing a method of achieving a limited form of device independent color. This standard is based on PostScript@ Level 2's ability to work with color spaces defined in terms of CIE 1931 (XYZ)-space and allows hardware and software to work in a wide array of device-dependent color color spaces while ensuring convertibility amongst spaces within the limitations of color space gamuts. Such a specification thus allows us to enjoy the efficiency of working in a device's native space without sacrificing data portability.

B

TWAIN 16/32 bit Thunker

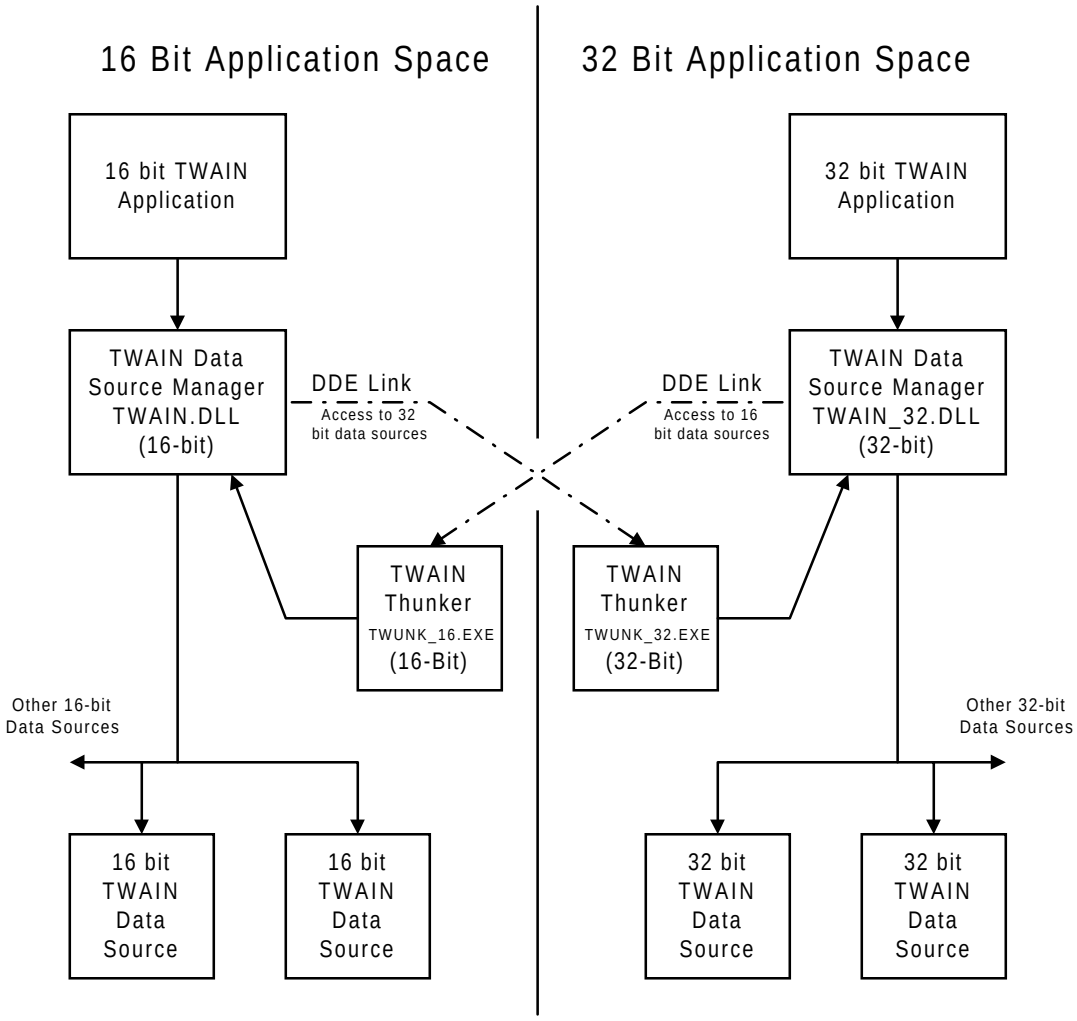
Chapter Contents

Description

Thunking is a special communication required to allow TWAIN applications compiled for 16-bit Windows 3.x to attach to TWAIN data-sources compiled for 32-bit Windows 95/NT and vice-versa. The TWAIN Thunker sits between an application and a Data Source.

As an example of Thunking, consider a 32-bit Windows '95 application accessing a 16-bit scanner data source. The 32-Bit application communicates to the TWAIN_32.DLL library. This DLL uses DDE commands to access TWUNK16.EXE. The EXE in turn uses Function Calls to call the 16-bit TWAIN.DLL, which calls the 16-bit data source.

TWAIN Thunking Solution



Thunking was developed to bridge the gap between 16-bit and 32-bit development.With the release of Windows 95, users who migrate from Windows 3.x find vendors of TWAIN applications and data-sources have been replacing their products with those compiled for Windows 95/NT. Unfortunately, a TWAIN application compiled for Windows 3.x can not communicate with a TWAIN data-source compiled for Windows 95/NT (or vice-versa) without assistance. Thunking provides the translation assistance that TWAIN applications and data-sources compiled for Windows 3.x require to communicate with the new generation of products compiled for Windows 95/NT.

There is an installed base of millions of image-acquisition devices providing TWAIN compliance under 16-bit Windows 3.x. An end-user who upgrades half of their TWAIN path will be unable to communicate through TWAIN without technical support. Without a TWAIN THUNKER this technical support is replacing the 'other half' of the TWAIN path with a Windows 95/NT component, if there is one available.

The THUNKER solution requires four pieces:

TWAIN.DLL
TWAIN_32.DLL
TWUNK_16.EXE
TWUNK_32.EXE

These four pieces must be installed in the WINDOWS subdirectory. All data-sources compiled for Windows 3.x will continue to be installed in the WINDOWS\TWAIN subdirectory. All data-sources compiled for Windows 95/NT must be installed in the WINDOWS\TWAIN_32 subdirectory.

TWUNK_XX.EXE is executed by the connection of the first application to the TWAINXX.DLL using the MSG_OPENDSM. The TWUNK_XX.EXE is effectively bound to the specified application and will continue to execute until the application disconnects from the TWAINXXX.DLL using the MSG_CLOSEDSM.

Any time a THUNKING application disconnects from the TWAINXXX.DLL, the TWUNK_XX.EXE will remain inactive until an application attempts to connect to the TWAINXXX.DLL using the MSG_OPENDSM.

What Limitations are Associated with THUNKING?

The following limitations are associated with THUNKING:

- The TWAIN THUNKER only supports ONE THUNKING session at a time. This session is allocated on a first-come-first-serve basis.
- The TWAIN THUNKER cannot support the following DSM_Entry parameters due to the variable or unknown size of the 6th parameter pData. We must know the exact size to pack information for DDE transfer. These parameters are currently trapped with return of TWRC_FAILURE/TWCC_BUMMER for a more 'graceful' failure:
  - ✓ DG_CUSTOMBASE
  - ✓ DAT_CUSTOMBASE
  - ✓ DAT_CIECOLOR
  - ✓ DAT_GRAYRESPONSE
  - ✓ DAT_RGBRESPONSE
  - ✓ DAT_JPEGCOMPRESSION

Note: We cannot use GlobalSize because it is possible (if not likely) that pData was not created with GlobalAlloc, in which case GlobalSize returns an ambiguous value.

The above limitations do not apply during TWAIN sessions that do not THUNK. These are 16 to 16 bit communication and 32 to 32 bit communication.

Glossary

ADF

Automatic Document Feeder.

Advanced-Level (Compliance)

Describes the implementation of an application or Source that has implemented the complete set of all defined Operations, Messages, and Capabilities for any particular Data Group.

API

Application Program Interface.

App

Short-hand for "application".

Basic-Level

Describes the set of Operations, Messages, and Capabilities that an application or Source is required to support to be TWAIN-compliant.

Buffer

Usually a contiguous block of main memory that is used to temporarily hold some part of a larger body of data. In our case, Buffered Memory data transfers from the Source to the application use "transfer buffers." The application is required to set up one or more transfer buffers, preferably of a size requested by the Source. The Source is told by the application where these buffers are in memory. The Source fills each buffer, one at a time, as data becomes available for transfer to the application. Whether these buffers are maintained by the application as the "permanent" repository for the transferred data or are filled by the Source, copied to a different location by the application, then disposed of, is wholly optional for the application.

CAP

An abbreviation for Capability. Also the prefix for Capability identifiers (CAP_xxxx).

Capability

A TWAIN identifier classification. Each capability is specified by a unique CAP_xxxx or ICAP_xxxx identifier. Each capability specifies a particular attribute of the Source with which it is associated. An application may inquire the current value of any capability supported by the Source, may inquire all the supported values for a capability, may attempt to set the current or available value(s) of the capability, or may reset the capability's value to its default. The process of inquiring and setting capability values is called "Capability Negotiation."

Capability Negotiation

A feature of the TWAIN protocol that allows an application to inquire the value or values of a control or setting of the Source and request that the control or setting be set to a particular value or set of values. This is how an application finds out, for instance, what resolution or resolutions a Source can acquire image data at. The application can then request the Source set itself to one of those values. This is, however, a negotiation. Robust Sources will comply with the application's request. Less capable (or less useful) Sources will fail on the application's request to set the value. Sources must never fail an inquiry, however.

Coalition

A group of individuals and companies who expressed interest in evaluating the work of the Working Group. The primary review body of the various drafts of the TWAIN Toolkit. The Coalition is open to any individual or organization who wishes to participate.

Code Resource

A Macintosh resource type that can be loaded by an application and executed as part of that application. The resource includes useful information like the location of the entry point(s) into the resource. The Source Manager and Sources on the Macintosh are implemented as code resources.

Condition Code

A secondary status or error code which can be requested by an application to get more specific information about the condition of the TWAIN session following many operations. The application will typically request this additional information in response to receiving certain Return Codes such as TWRC_FAILURE.

Custom-Level

A set of Data Argument Types, Messages, and Capabilities that are defined and specified by the manufacturer and not by the specifications in this Toolkit. DATs, CAPs, and MSGs all allow for custom-definitions that represent functionality particular to a vendor's product(s) or to act as temporary extensions to the formally defined identifiers. These identifiers are readily sensed by the application or Source since the identifier's high-bit (bit 15) is set.

DAT

An abbreviation for Data Argument Type. Also the prefix for Data Argument Type identifiers (DAT_xxxx).

Data Argument Type

(Also DAT) Used in an operation triplet to specify the "type" of the data referenced by the pData parameter of DSM_Entry( ) (and, for Sources, DS_Entry( )). A DAT may specify a data structure (a unique DAT references each defined structure) or an individual value. Knowing the DAT tells the application, Source Manager, or Source the precise form of the data to be acted upon.

Data Group

A Data Group, or DG, identifies the general category of the operation specified by the parameter list of the DSM_Entry( ) call. There are currently only two categories defined. The first, DG_CONTROL, indicates that the operation affects the flow or character of the TWAIN session (such as opening or closing a Source or the Source Manager). The second, DG_IMAGE, says the operation affects some aspect of the image acquisition, the image data, or the Source's image controls/capabilities.

Data Source

Another name for a "Source."

Data Source Manager

Another name for the "Source Manager."

DC

An abbreviation for "Direct Connect."

DG

An abbreviation for Data Group. Also the prefix for Data Group identifiers (DG_xxxx).

DLL

An abbreviation for Dynamic Link Library. An MS Windows implementation that allows a block of code to be written and compiled separately from the application that will use the code. Both the application and the DLL are compiled in such a way that references to external variables belonging to the DLL are left unresolved at compile and link time. The application is compiled and linked with knowledge of the DLL's entry point(s) but no knowledge of the DLL's code addresses. At run-time, the application can make a call to the DLL's entry point. At that time, the DLL is made available to the application and all the references to the DLL in the application are resolved—hence the term "dynamic linking". The Source Manager and Sources under MS Windows are implemented in this fashion.

DS

An abbreviation for Data Source. Data Sources are now called "Sources."

DSM

An abbreviation for **D**ata **S**ource **M**anager. The Data Source Manager is now called the "Source Manager."

Element, TWAIN

An application, the Source Manager, or a Source. The actual coded entity being executed, referenced, or residing in storage.

Entry Point

The location of an externally available function in an executable file where the Program Counter will point when the function is called. There is only one TWAIN-defined entry point that is called by an application: DSM_Entry( ). There is only one TWAIN-defined entry point that a Source must provide for the Source Manager: DS_Entry( ).

Handle

A reference to a relocatable data item located in memory. On the Macintosh, a handle is a doubly-indirected reference to a data item in memory. Also known as a pointer to a pointer (a pointer to the address of the data item). Under MS Windows, a handle is a 16-bit index into a table managed by MS Windows. The entries in this table are themselves pointers to data items in memory. These indexes cannot be directly dereferenced by the application to access the data item. UNIX has no concept of a handle; its pointers are inherently relocatable.

ICAP

An abbreviation for Image-related CAPabilities. See Capabilities.

Message

A Message, or MSG, is used in an Operation Triplet to specify what action is to be taken on a particular data item or, for a data-less operation, what action is to be performed.

Modal (Source User Interface)

The Source tells the application that its user interface cannot lose the focus as long as it is visible. All events are directed to the Source while its user interface is presented. The Source's UI cannot be switched out of focus. The application will not be able to process events while the Source's UI is presented. This mode is appropriate for simple Sources whose UI is transient and requires little user interaction to perform a successful acquisition.

Modaless (Source User Interface)

The Source tells the application that its user interface can lose the focus while it is visible. The application must pass all events to the Source, which processes all events that belong to it. The application can process events while the Source's UI is presented. This mode is appropriate for most Sources. Such a Source may have more than one window and allows the user more flexibility and control than most modal UIs. Most Sources should strive to implement their user interfaces as modaless.

MSG

An abbreviation for Message. Also the prefix for Message identifiers (MSG_xxxx).

Native (Transfer)

A transfer mechanism and data format for acquiring data from a Source. This is the simplest transfer mechanism to implement.

This mechanism is best for applications that don't plan to use large blocks of data (especially large images) and need to have the operating environment provide the services to render the data to the screen and printer. The Source creates a RAM-based block into which is written the data in a format that is "native" to the host computer—PICT for the Macintosh, DIB under MS Windows. This transfer mechanism is limited in size to the amount of main memory that can be allocated in a single block within the run-time environment of the Source.

Operation

A specific action to be taken upon a particular set of data. Such operations are uniquely specified by an Operation Triplet. Execution of any particular operation is requested by the application by including the required triplet as three of the parameters in the Source Manager's entry point.

Operation Triplet

The complete name for a "Triplet". A combination of Data Group, Data Argument Type, and Message which uniquely specifies an Operation. That is, a specific action to be taken upon a particular data item.

Protocol

The rules and language of interaction for two TWAIN entities. The "language" is defined by the Operation Triplets which, in conjunction with the data upon which they act, define every action that can occur within TWAIN. The ability for a vendor to define custom Operations allows for specialized extension of this protocol. The TWAIN protocol is further defined by the rules and guidelines for how an entity should behave and the user interface guidelines for an application's UI. The API differs from the protocol in that the API is the portal through which TWAIN functions are accessed.

Return Code

The value returned from every call to DSM_Entry( ) (or DS_Entry( ) for Sources). This value indicates that the call performed successfully, failed, or achieved some other specific condition. The possible values are listed for each operation in Chapter 7. Certain Return Codes prompt the application to inquire for a more explicit Condition Code.

Session

The period while a TWAIN element (Source Manager or a Source) is connected to an application. Used to indicate when the TWAIN protocol is "active" between an application and the Source Manager or a Source—there is a Source Manager session as long as the application has the Source Manager open, and another session for each Source the application opens.

Shared Memory

An area of main storage that can be accessed (blocks of memory can be allocated and deallocated) by the application(s), Source Manager(s), and Source(s) during any given session.

Source

A TWAIN entity whose purpose is to provide data to a TWAIN-compliant application. A Source typically controls a peripheral device, though a virtual device easily fits the model for a Source. The actual device may be locally connected to the host computer or may be remotely connected over some network. The Source abstracts both the device and the method of hardware connection (SCSI, serial, etc.) from the consuming application. The application can deal with every Source in exactly the same programmatic way, if it so chooses. Every Source has built into it a controlling user interface that appears within every application that utilizes the Source (except for the few applications that suppress the Source's UI or replace it with one of their own). This built-in UI will typically be the best way for users to interact with the Source from their application. A Source is wholly subservient to the consuming application and always tries to provide the type of data, in the desired format, and with the characteristics and attributes the application explicitly or implicitly requests. A Source is implemented as a code resource on the Macintosh and a DLL under MS Windows. Under Windows, only one copy of a Source will be active in memory at any given time. On the Mac, there will be a separate copy of a Source for each application accessing it (assuming that the Source supports connection to multiple applications simultaneously).

Source Manager

The "traffic cop" of the TWAIN protocol. The Source Manager is located between the application and the Source(s) it is connected to. This logical connection flows through the Source Manager with no Operation Triplets passing directly to the Source from the application. Even though the actual data transferred from the Source to the application bypasses the Source Manager, the protocol handshaking the data still routes through the Source Manager. The Source Manager manages all the connections between application and Source. The Source Manager allows the user to select the Source to be used for an Acquire. It is responsible for loading and unloading the selected Source and for assuring that all information for any particular Source from any particular application is correctly routed. The Source Manager is provided to developers free of charge by the Working Group. The Source Manager is implemented as a code resource on the Macintosh and a DLL under MS Windows. Under Windows, there will be only one copy of the Source Manager active in memory at any given time. On the Mac, there will be a separate copy of the Source Manager for each application accessing it.

State(s)

TWAIN defines seven unique states, or conditions, that every TWAIN session moves through, and that the TWAIN entities occupy. The first state exists prior to initiation of the TWAIN session by the application. The next two involve only the application and the Source Manager—no Sources. The last four revolve around setting up the Source(s) to acquire data and the data transfer itself. During these four states, the Source Manager simply passes through the communications between the application and Source(s). While a TWAIN entity occupies a particular state, it will continue to do so until the Operation that transitioned it into that state completes, or until a new Operation is received that specifically forces another state transition (in the forward or reverse direction).

State Diagram,
State Transition Diagram

A graphical representation of the seven TWAIN session states and the forward and reverse state-to-state transitions. This diagram (see Chapters 2 and 3) is very important to understanding the control flow of any TWAIN session.

Strips

A specific form/organization of image data, typically a bitmap. A rectangular block of image data that encompasses the entire perceived "width" of the acquired image. The block will usually only contain a contiguous portion of the image's "height." For example, take an 8-1/2 x 11" (or A4, or whatever) piece of paper. Orient it as a "portrait" (taller than wide). Fold the paper so that it gets shorter. Fold it a few more times. Open the paper and look at it in "portrait mode" again. If you pretend that the page represents an image's bitmap (rectangular, of course) each of the rectangular areas bounded by folds represents a strip. File formats, like TIFF, and applications that run on machines with non-infinite amounts of RAM prefer to (or must) deal with image data in these strips. (See also Tiles)

Tiles

The generalized form of a Strip (see the definition, above). The difference is that a tile may be narrower than the image's "width." The width will usually be divided into an integer number of tiles.

Toolkit, TWAIN

This set of documents and associated software. Describes and specifies the TWAIN protocol, API, implementation rules and guidelines, and related source, object, and executable software.

Triplet

A contracted version of the phrase Operation Triplet.

UI

Shorthand for "user interface."

Working Group, TWAIN

An *ad hoc* group of companies whose key role is to create and maintain an open specification for generalized interchange of raster information from source to consumer. The companies assembled represent a relatively broad base of imaging expertise for both software and hardware products. Other interested parties contributed through the Coalition.