DataEXPORT/DLL™ Developer's ToolKit

User's Guide

User’s Guide to:

DataExport Version 6

By Spalding Software, Inc.

Contents

Chapter 1: Installation

This chapter describes how to install DataExport/DLL on a single computer for development purposes. Experienced PC users should read the **QuickStart** section to understand the essentials of installing the ToolKit.

Installing the DataExport/DLL Developer’s ToolKit

DataExport/DLL requires an IBM or compatible PC running Windows 3.1 or above with a minimum of 4MB RAM and 4MB of hard disk space available.

If you are an experienced PC user, you will probably want to use the instructions in the **QuickStart** section when installing DataExport/DLL. If you are new to Windows, or if you do not understand the instructions given in the **QuickStart** section, read the **Step by Step** section and follow the outlined procedure to load DataExport/DLL.

QuickStart

Insert Disk 1 into the A: drive. Run the Setup program by switching to the Program Manager, choosing File Run, typing “A:SETUP” and pressing OK. Follow the instructions in the Setup program.

The installation program will copy the required files into the directory you specify. The redistributable DLL files will be installed to your Windows system directory. See Appendix C and the README.TXT file for more information on the DataExport/DLL distributable files.

To use DataExport/DLL with your Windows application, you must first code for the output of data from your application. You can pass data to the DLL using either its Cell-by-Cell API or by using the File-Based Translation method. Refer to Chapter 2 for further information.

Step by Step Installation

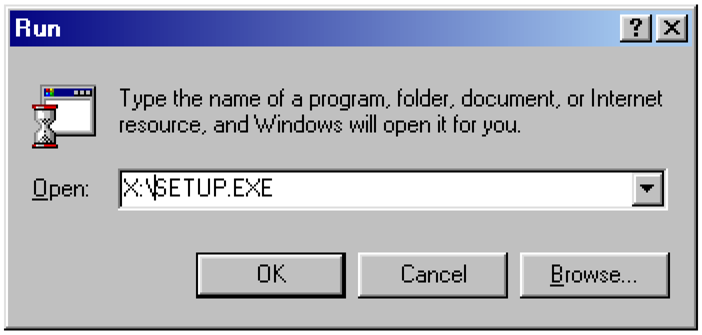
To use DataExport/DLL, you must first install the program on your hard drive using the supplied installation program called SETUP. This program walks you through the installation procedure by asking you where you want to install the program files, copying the program files to your hard drive and creating a new program group.

The steps below explain the procedure for using the Setup utility that is provided with the DataExport/DLL installation disks. Follow the procedures below to install the ToolKit on a single computer.

Note: No other programs other than Program Manager or File Manager should be running during installation. Exit all other applications before installing DataExport/DLL.

Procedure:

1. Switch to the Windows Program Manager.
2. Insert DataExport/DLL CD-ROM into the drive.
3. From the **File** menu, choose **Run...**
4. The Run dialog box appears. Type the letter of the disk drive and then ":SETUP".


Run
?
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open:
X:\SETUP.EXE
OK
Cancel
Browse...

Running the Setup program for DataExport/DLL

5. Press the OK button to run Setup.

The Setup program initializes and then the DataExport/DLL Setup screen appears. The first dialog box warns that no other applications—other than Program Manager or File Manager—should be running.

6. Press OK to continue
7. To accept the default directory and install DataExport/DLL, press the Continue button. If you want to install DataExport/DLL to a different directory, type in the new directory and then press Continue.

   The installation begins and a dialog box indicates the progress of the Setup program.

8. If necessary, insert additional installation disks as prompted by the Setup program. If the Setup program asks for installation disks which you do not have, check the original packaging before contacting Spalding Software.

   After copying files, the Setup program will build a DataExport/DLL program group and notify you upon completion.

Note: The Setup program writes a log of the installation process called INSTALL.LOG to the directory where DataExport/DLL is located. This log lists what files were copied to your hard disk and where the files are located. Keep this file as a record in case you or your system administrator needs to uninstall DataExport/DLL.

Congratulations! You have successfully completed the installation of DataExport/DLL! Be sure to examine the README.TXT file which is discussed in the next section.

The README.TXT File

The DataExport/DLL diskettes may contain some new information not yet added to this manual. This information will be in a file named README.TXT. Please read this file to get the latest information about your version of the DLL. If this file does not exist, don't worry; it simply means that this manual is completely up to date.

To view README.TXT information, go to the Program Manager and in the DataExport/DLL program group, double click on the README.TXT icon. The text will be loaded into the Windows Notepad application. You can also view the file with any standard Windows word processor.

Technical Support

Spalding Software provides technical support for DataExport/DLL to help you solve problems with installation and use of the program. If you have questions or problems not addressed by this manual or the online help, your next best source of information is our online information centers on the World Wide Web:

website:	http://spaldingsoftware.com
email:	dx-support@spaldingsoftware.com

If you cannot resolve the problem through the above methods, Spalding Software provides telephone support for DataExport/DLL to help you solve problems using the library. Before calling for support:

- Try to duplicate the problem, step by step, to see exactly what happened and when the problem occurred.
- Be at your computer when you call. Have your manual and license number handy.

Telephone support is available from 9 AM to 5 PM US Eastern Time (GMT -5) on normal business days at:

+770-449-0594 voice

+770-449-0052 fax

Chapter 2: Overview

This chapter explains the general purpose, benefits and use of the DataExport/DLL Developer's ToolKit. Spreadsheet and database terminology used in this manual is also covered here, along with decisions to be made when implementing the DLL in your application.

DataExport/DLL Developer's ToolKit

The DataExport/DLL Developer's ToolKit is a set of Dynamic Link Libraries (DLLs) for Microsoft Windows applications. The ToolKit was created to provide developers with a fast, easy, reliable way of exporting data from their applications into multiple file formats—without the hassles of analyzing and coding for each format. DataExport/DLL is designed to provide the code behind a File Export command in your program.

Benefits

Low Learning Curve - DataExport/DLL allows you to output to multiple data formats without requiring you to know the complexities of those formats. A simple, uniform set of methods allows you to specify an output type and pass off your data. DataExport/DLL takes care of the rest.

Complete Solution - At the writing of this manual, DataExport/DLL provides over 40 output formats, including the latest versions of Excel, Access and Paradox. DataExport/DLL also covers the new and old standards, like Lotus WKS, CSV, DIF and dBase.

Customer Satisfaction - DataExport/DLL provides an immediate and complete resolution to this often-heard question: "Can your software output to XXX format files?" Incorporating DataExport/DLL in your product means you can instantly satisfy demands for output types in all major data formats. DataExport/DLL also allows you to add impressive integration with database and spreadsheet applications.

Inter-Application Integration - In addition to creating new data files, DataExport/DLL can also append data to existing data files, allowing you to achieve further integration with other applications. With DataExport/DLL you can add data or results from your program to existing databases and spreadsheets, thereby increasing communication, saving time and improving the workflow of your customer.

Output Formats

DataExport/DLL provides over 40 spreadsheet and database output formats. These output formats allow for support of many more applications and versions than those listed below, since many programs—especially database management apps—share one of these common file formats. DataExport/DLL outputs formats used by these major applications:

Application	DLL FileType
Access 1.1	MDB1
Access 2.0	MDB
Access 3.0	MDB3
Access 4.0	MDB4
Alpha 4	DBF3
Approach	DBF3
Clarion	DAT
dBase II	DBF2
dBase III	DBF3
dBase IV	DBF4
Excel version 2.1	XLS
Excel version 3.0	XLS3
Excel version 4.0	XLS4
Excel version 5.0	XLS5
Excel version 97/2000/XP	XLS8
FoxPro	DBF3
Lotus 1-2-3 release 1 and 1A	WKS
Lotus 1-2-3 release 2.x	WK1
Lotus 1-2-3 release 3.x	WK3
Lotus 1-2-3 release 4.x and 5.x	WK4
Microsoft Word data document	WRD
Paradox 3.5	DB35
Paradox 4.0	DB
Quattro	WKQ
Quattro Pro	WQ1
Quattro Pro 5.0 for Windows	WB1
SYLK Multiplan	SLK
Symphony release 1.0	WRK
Symphony release 1.1, 1.2 and 2.x	WR1
Word Perfect 5.0 secondary merge file	W50
Word Perfect 5.1 secondary merge file	W51

Spreadsheet and Database Terminology

The terminology of database and spreadsheet structures can be somewhat confusing at times. People in the world of computers talk about “columns, cells, rows and sheets” or “headers, fields, records and tables” depending on the kind of application they use most frequently. At Spalding Software, we tend to use all these terms and interchange them depending on the phase of the moon and who made the coffee that morning.

Since the structure of a data file is of crucial importance to you, your end-user and DataExport/DLL, it is absolutely critical that you are clear about what these terms mean and their relationship to each other:

Cell - In the spreadsheet world, a cell refers to a single unit of data, for instance, a single dollar amount, a street address, or a zip code. In a database, a single unit of data is sometimes called a *field*. However, a field can also refer to a *column* of data (see below).

Row - A horizontal line of cells containing data of different types, which usually constitutes a set of data, for example, a line of cells containing a company name, street address, city, state, zip code and phone number. In the database world, this set of data is called a *record*.

Column - A vertical array of cells which usually contain data of the same type, for instance, a list of companies or a list of phone numbers. In a database, a column of data is sometimes referred to as a *field*.

Sheet - A collection of columns and rows which usually constitutes a complete set of data, for example, the contact information for customer companies or the invoice log for a sales department. Some spreadsheet formats can have multiple sheets per file. In the database world, a sheet is structurally similar to a *table* in a database file.

Field - In the database world, a field either refers to a single unit of data—like a part number—or a vertical set of data of the same type (e.g., a list of part numbers). In the spreadsheet world, a field is equivalent to either a *cell* or a *column* of data, depending on how it is used.

Record - A horizontal line of fields containing data of different types, which usually constitutes a set of data, for example, a line of fields containing a company name, street address, city, state, zip code and phone numbers. In a spreadsheet, this set of data is called a *row*.

Header - A structure at the beginning of a database file that contains information about the file, including the definition of the type of data contained in each field in the database. Although there is no equivalent structure to a header in a spreadsheet file, a header can be thought of as a structure that contains *column definitions*. This term is used in the DataExport/DLL manual as a substitute for *field definitions*.

Table - A structure containing information that usually constitutes a complete set of data, for instance, contact information for customer companies or the log of invoices from a sales department. In dBase, a single file constitutes a table, while an Access database file can contain multiple tables. In the spreadsheet world, a table is structurally similar to a *sheet*.

This manual standardizes on the spreadsheet terms for data structures—cells, columns and rows—for the sake of clarity. This decision requires the use of a new term for the definition of database field, which is called a *column definition* in this manual.

Implementation Decisions

There are a two primary decisions that you must make when planning the implementation of DataExport/DLL in your Windows application.

1. *What programming language should I use?*
2. *Should I use the DataExport/DLL methods or intermediate disk files to output data to different formats?*

The first question should be easy to answer. The only requirement for the language you use is that it must allow you to call the functions of an ActiveX DLL. Any language will do as long as it meets this requirement.

The answer to the second question is slightly more involved. DataExport/DLL allows you to output file formats in two different ways:

a. Cell-by-Cell output, using the DataExport/DLL methods

b. File-Based translation, using a raw ASCII data file and an Export Definition File

Cell-by-Cell output involves communicating the data you want to output through the functions of the DataExport/DLL methods. We assume that this option is preferable to most programmers, since it offers the most control and does not involve any intermediate, temporary files.

File-Based translation requires writing two temporary files—a data file and a translation control file, or EDF—to disk and then making a single method call to DataExport/DLL to initiate translation. This option may be preferable if you wish to quickly adapt an existing output function—which outputs a CSV or other text file—to output other formats.

Note: For developers using DataExport/DLL as a substitute for the translation capabilities of DataImport, the File-Based translation method will most likely be your choice. The Export Definition File (EDF) can provide the functions you need for report or text file translation (See Appendix B for more information). A pre-defined DataImport for Windows Mask file (MSK) can also be used in place of an EDF.

If you need to know more about the process of translation with DataExport/DLL, please continue reviewing this chapter. Otherwise, you should jump to Chapter 3 and review “Building a User Interface for Selecting Output Formats” section to find out what user interface is needed for implementing the functions of the DLL.

DataExport/DLL Process Flow

The process of implementing DataExport/DLL in your application depends upon how you use it. The DLL can be implemented with Cell-by-Cell output or through File-Based translation using intermediate files containing instructions and raw data in ASCII text format.

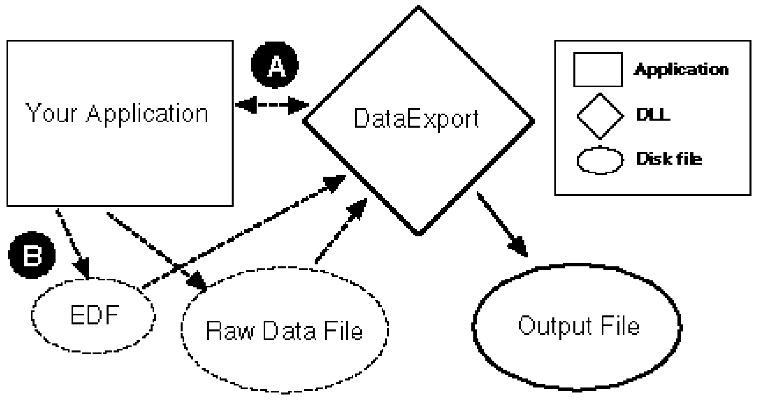


Diagram of DataExport/DLL data translation and output process.

The diagram above illustrates the data translation process of DataExport/DLL and two possible implementations:

A) The Cell-by-Cell (or field-by-field) output method, which uses DataExport/DLL methods to initialize an output file, pass data to the file and close it.

B) The File-Based Translation method, which uses an Export Definition File (EDF) to define the final output format, a Raw Data File as a data source and a single method call to the DataExport/DLL to perform the translation.

Cell-by-Cell Output Method

The Cell-by-Cell output method uses the DataExport/DLL methods to establish a file output format and then communicate each unit of data to the DataExport/DLL. This process is called Cell-by Cell output because it requires calling the DLL once for each cell—or field in a database—that is output.

Information written to the buffer is retained until overwritten with `DXPut's` or the buffer is cleared with `DXClearBuffer`.

In a Cell-by-Cell implementation, the calling application begins translation using the `DXInitTrans` method. The data format of each column (or field) is then specified with `DXDefData`. The calling application then begins writing data for a row (or record) to a buffer with the `DXPut` commands, completing each row with the `DXWriteBuffer` command.

The process of passing data with the `DXPut` commands and writing the buffer continues until all rows, or records, have been written. The calling application then completes the output file with the `DXCloseFile` command.

File-Based Translation Method

The File-Based Translation method uses an Export Definition File (EDF) to define an output format, a Raw Data File to communicate the data and a single method call to initialize translation. This method of output uses the DataExport/DLL as a kind of batch processor.

In a File-Based Translation implementation, the calling application creates an EDF containing information about the type of output file that is desired, the column definitions (or field definitions) and names of the files. The application also writes a Raw Data File in a simple ASCII format to be processed by DataExport/DLL. Finally, the calling application issues a `DXLaunchEngine` call to begin translation.

File-Based Translation Method Using a DataImport Mask

The MSK file format is a proprietary technology of Spalding Software and must be created using the DataImport program.

The File-Based Translation method may also be used with a DataImport for Windows Mask file (MSK) substituted for the EDF and, optionally, a report file or other text file may be substituted for the Raw Data File. Developers wanting to use this method should develop their Mask using DataImport and then follow the instructions for the File-Based method, substituting an MSK for the EDF and, optionally, a text file for the Raw Data File.

Refer to your DataImport documentation for more information on how to create a Mask file for your report or other text file.

DataExport/DLL Methods

In order to use the DataExport/DLL methods, you must make a reference to the DLL in your program. This section provides a brief review of the methods and their purpose. Refer to Appendix A for specific syntax and prototypes for C and Visual Basic.

Querying the DLL for Supported Formats

The DataExport/DLL provides two methods which allow you to query for the number and types of formats it currently supports. These functions are intended for dynamic construction of dialog boxes and other communication with your program.

`DXGetNumTypesW`	Returns the number of supported output types in Unicode strings
`DXGetNumTypesA`	Returns the number of supported output types in ASCIIZ strings
`DXEnumTypesW`	Returns output type information for a `FileType`, including format name, file extension, type of file format and whether a Sheet or Table name is required in Unicode strings
`DXEnumTypesA`	Returns output type information for a `FileType`, including format name, file extension, type of file format and whether a Sheet or Table name is required in ASCIIZ strings

Cell-by-Cell Output Functions

There are ten methods used in a Cell-by-Cell output implementation:

`DXInitTransW`	Initialize an output file with Unicode parameters
`DXInitTransA`	Initialize an output file with ASCIIZ parameters
`DXDefDataW`	Define a data format for each column with Unicode parameters
`DXDefDataA`	Define a data format for each column with ASCIIZ parameters
`DXPutInt`	Write a 2 byte signed integer to buffer
`DXPutLong`	Write a 4 byte signed integer to buffer
`DXPutSingle`	Write a 4 byte single precision IEEE Real to buffer

`DXPutDouble`	Write a 8 byte double precision IEEE Real to buffer
`DXPutStringW`	Write a Unicode string to buffer
`DXPutStringA`	Write a ASCIIZ string to buffer
`DXWriteBuffer`	Write buffer to output file
`DXClearBuffer`	Clear data from buffer
`DXCloseFile`	Close output file

Refer to Chapter 3 and Appendix A for more information about using these methods.

File-Based Translation Functions

There is only a single method for initialization of a File-Based Translation. This function is similar to the command line call of another of our products: DataImport:

`DXLaunchEngineW`	Run translation with specified Unicode parameters
`DXLaunchEngineA`	Run translation with specified ASCIIZ parameters

Refer to Chapter 3 and Appendix A for more information about using this API function.

Chapter 3: Using DataExport/DLL

There are two distinct ways of implementing DataExport/DLL in your application: Cell-by-Cell output, which uses the DataExport/DLL methods to define and then write data into an output file and File-Based translation, which uses a raw data file and an Export Definition File (EDF)—or a DataImport Mask (MSK)—to create an output file. This chapter explains each of these output methods, as well as the user interface requirements for use of the DLL.

Building a User Interface for Selecting Output Formats

Unless you are using the DLL for purely internal functions, some user interface will be needed in order to implement the functions of DataExport/DLL. The DLL does not provide a user interface—you must construct an interface in the style of your application.

DataExport/DLL allows you to query and retrieve information about the supported output formats and their characteristics. We recommend using this resource to dynamically build your dialog boxes at run-time, rather than hard-coding them. This implementation will allow you to easily accommodate new output formats as they are added to the DataExport/DLL. If you automate the interface construction and communication of data properly, adding new formats to your application should be as simple as dropping a new version into your program's folder.

Obtaining DataExport/DLL Info and User Input

Before you start outputting file formats with DataExport/DLL , there are a few questions you need to answer:

1. *What file formats does my version of the DataExport/DLL support and what are the characteristics of those formats?*
2. *What is the format, filename and path for the output file?*

The first question can be answered by DataExport/DLL itself using the `DXGetNumTypes` and `DXEnumTypes` methods. Assuming you have written a routine to dynamically construct a dialog box for your user, you need to obtain this information to setup your dialog(s).

The second question is then answered through interaction of your user with the dialog box you have assembled from the information obtained in the previous step.

Once the user has provided the required input, you are ready to start outputting a new data file.

Querying the DLL for Supported Output Formats

DataExport/DLL provides two methods which allow you to query for the number and types of formats it currently supports. These methods allow you to dynamically construct a File Export dialog box at runtime.

`DXGetNumTypes`	Returns number of supported output types
`DXEnumTypes`	Returns pointers to information about a specified `FileType`, including format name, file extension, type of file format and whether a Sheet or Table name is required

In order to use this resource, simply query the component when your application loads or when a user selects the File Export command (or equivalent) and build the output file list and information for format-conditional input variables (e.g., TableName requirements)

Code Examples: Querying for Output Formats

These sample routines get the number of types in the DataExport/DLL and populate arrays with information about what types are available. These arrays could be used to populate a dialog and provide a pull-down list for the user.

Visual Basic Example: Using `DXGetNumTypes` and `DXEnumTypes`

```
Sub GetTransTypes ()

    Dim I&, TypeFlag&, NameFlag&, MaxTypes&, RetVal&

    Dim DX as DataExport

    Set DX = New DataExport


    ' get the number of translation types available
    MaxTypes = DX.DXGetNumTypes()


    ' dimension the arrays that will hold information
    ' about the types

    ReDim TypeExt$(MaxTypes)

    ReDim TypeName$(MaxTypes)

    ReDim TypeofTrans$(MaxTypes)


    ' loop through all types and get the information
    For I = 1 To MaxTypes

        RetVal = DX.DXEnumTypesW(I, _⇔
                     TypeName(I), _⇔
                     TypeExt(I), _⇔
                     TypeofTrans(I), _⇔
                     TypeFlag, _⇔
                     NameFlag)

    Next I
```

```
End Sub
```

C Example: Using `DXGetNumTypes` and `DXEnumTypes`

```
#include "dxtrans.h"
int GetTransTypes (void)
{
    /* set up the buffers for returned information */
    char NameBuff[40];
    char ExtBuff[4];
    char TypeBuff[10];
    int MaxTypes;
    int TransType;
    int NameNeeded;
    int i;


    /* get the number of translation types in DLL */
    MaxTypes = DXGetNumTypes();


    for (i = 1; i <= MaxTypes; i++) {
        result = DXEnumTypes(i, NameBuff, ExtBuff,
                 TypeBuff, &TransType, &NameNeeded);
        /* store the returned values into arrays or
           other structures */
    }
}
```

The ⬄ symbol in the Visual Basic examples indicates a continuation of the same line appears on the next line.

See Appendix A for the command syntax and prototypes for C and Visual Basic for these output format query functions.

Note: These routines do not create an array with the information for the `NameNeeded` parameter of the `DXEnumTypes` function (This parameter is only relevant for Access, Excel 5.0 and other output file types that use names for sheets or database tables). In your code, you should either have the user supply a name, assign a default name in your application or allow DataExport/DLL to use its default sheet and table names, “Sheet1” and “Table1”, respectively, to support these formats.

With the array values obtained from these routines, you can build dialog boxes or store this information for future use. The next step in this process is to obtain user input about the name and type of output file that is desired.

Required User-Defined Variables

You will typically need a dialog box where the user can specify these minimum parameters:

1. Output file format (Access, Excel, dBase, etc.)
2. Filename and path

Note that Access and Excel 5.0 formats can require additional information. For example, for Access output, you can specify the name of a Table you will be writing in the database file (DataExport/DLL uses a default name of “Table1” if none is specified).

In order to accommodate the different requirements for each file format, we suggest creating a dynamic dialog box which displays or removes input options depending upon the type of output file selected. The following input variables should be provided in this dialog box when the indicated format is chosen:

Table Name - for Access databases

Sheet Name - For Excel 5.0 spreadsheets

If no names are specified DataExport/DLL provides a default name of “Table1” for Access tables and “Sheet1” for Excel 5.0 sheets. See Chapter 4 for more information on output file formats.

Cell-by-Cell Output

Cell-by-Cell output is performed through the DataExport/DLL, by first initializing an output file, defining the columns/fields in the output file, writing data to a buffer one row/record at a time and then closing the output file. This section further explains the implementation of the Cell-by-Cell output method.

Methods

There are ten Methods used in a Cell-by-Cell output implementation:

`DXInitTrans*`	Initialize an output file
`DXDefData*`	Define data format for each column
`DXPutInt`	Write a 2 byte signed integer to buffer
`DXPutLong`	Write a 4 byte signed integer to buffer
`DXPutSingle`	Write a 4 byte single precision IEEE Real to buffer
`DXPutDouble`	Write a 8 byte double precision IEEE Real to buffer
`DXPutString`	Write a zero delimited string to buffer
`DXWriteBuffer*`	Write buffer to output file
`DXClearBuffer`	Clear data from buffer
`DXCloseFile*`	Close output file

* These functions *must* be used to create an output file when using Cell-by-Cell output with DataExport/DLL.

Refer to Appendix A for specific syntax and prototypes for C and Visual Basic. The following sections explain the use of these functions through an example implementation of DataExport/DLL . These examples assume that you have properly declared these functions as detailed in Appendix A.

Opening an Output File

DataExport/DLL can create a new output file or write to an existing file. In either case, outputting to a data file always begins with the `DXInitTrans` function call.

```
DXInitTrans(FileName, FileType, TableName, NumCol,
    IfExistFlag, handle)
```

This function returns a `handle` that is used with all subsequent DX calls and establishes the fundamental structure of the output file, including the file name, the file type, sheet/table name (if required) and number of columns (or fields). The `DXInitTrans` command also specifies whether to overwrite or append data if the specified output file exists (`IfExistFlag`).

Note: If you are writing data to an existing file, you *must* use the format of that file in your output definitions.

Declaring Column/Field Definitions

Once the output file has been opened with the `DXInitTrans` command, the next step is to establish the column definitions, or field types. Column definitions are an exceedingly important consideration when outputting to a database file, since writing the wrong kind of data into a field can cause serious errors in the database.

```
DXDefData (handle, Index, DataType, DataWidth,
    DataDecimal, DataName)
```

One `DXDefData` call per column is required (e.g., for a 5 column/field output file `DXDefData` is called 5 times). Each call specifies the type of data within the column being defined (`DataType`), the character width of all cells within that column (`DataWidth`), any implied decimals and the name of the column/field.

The `DataType` parameter is particularly important, since it defines what kind of data will be accepted by DataExport/DLL when you write data to the buffer with the `DXPut` commands. The primary distinction between data types is whether the data is numeric or an alphanumeric string. `DataTypes` are further distinguished between the size of the string and ‘special’ values, specifically date and time values. See “Writing Cells in a Row/Record” below and “DataTypes” in Appendix A for more information.

The `DataWidth` is also an important parameter since it defines the maximum number of characters that will be accepted in a cell or column. DataExport/DLL will truncate strings that exceed the maximum defined width of a column when outputting to databases, SDF and fixed field files. Proper use of the `DataWidth` parameter requires that you know the maximum character width of any cell you will write to a column.

Code Example: Opening and Defining an Output File

The following routines initiate an dBase IV output file and define three columns/fields in the output file.

Visual Basic Example: Opening and defining an output file with the `DXInitTrans` and `DXDefData` functions

```
Sub DoCellTrans (ByVal DX as DataExport)

    ' This partial routine initializes a new output
    ' file and defines three columns.


    Dim FileName$, FileType$, hTable&, Result&


    ' FileName and FileType could easily be passed
    ' from the main program, as well as the data

    FileName$ = "c:\dx\demtext.dbf"

    FileType$ = "DBF4"


    ' start up the translation engine and get back
    ' the hTable

    result = DX.DXInitTransW(FileName, FileType, _⇔
               "", 3, 1, hTable)


    ' define the columns/fields of the data

    Result = DX.DXDefDataW(hTable, 1, _⇔
                DXData_Date, 8, 0, "DATE")

    Result = DX.DXDefDataW(hTable, 2, _⇔
                DXData_Text, 12, 0, "CITY")

    Result = DX.DXDefDataW(hTable, 3, _⇔
                DXData_General, 12, 3, "UNITS")


    ' clear out the buffer

    ' write rows/records (DXPut functions)

    ' close output file (DXCloseFile)

End Sub
```

See Appendix A for the syntax these output format query methods.

Note that the preceding examples do not show writing data to the DataExport/DLL buffer. This procedure is covered in the next example.

Initializing and Clearing the Row/Record Buffer

After the output file type and structure have been established, the `DXClearBuffer` function is used to initialize and clear out the DataExport/DLL buffer.

```
DXClearBuffer(handle)
```

This buffer is essentially a chalk board where the data for the output file is written. The buffer has a numbered cell for each column/field defined by the previous `DXDefData` functions. The buffer can hold one row/record at a time, so `DXPut` commands must be followed by a `DXWriteBuffer` to complete a row/record and commit it to the output file before writing a new row/record.

The DataExport/DLL buffer also *remembers* the last row/record that was written to it, so the previously written row information remains in the buffer until it is overwritten or cleared out. The `DXPut` functions overwrite any data currently in the specified cell—essentially wiping the cell clean before writing the new information. The `DXClearBuffer` command, however, can be used to wipe out all information in all cells in the buffer.

The `DXClearBuffer` function is only required to initialize the buffer and clear it out before beginning writing rows/records in a new output file. The command is not required in any subsequent action on that file. See Appendix A for syntax and more information.

Writing Cells in a Row/Record

After establishing column/field definitions using the `DXDefData` function, you can then begin writing data to the new output file using the `DXPut` functions. One `DXPut` function is used for each cell/field that is written to the buffer; so, for a 5 column output file, five `DXPut` calls are made to write data for that row/record. Each row/record is written separately. The `DXPut` commands are listed below:

`DXPutInt(handle, Index, Value)`
Write a 2 byte signed integer to buffer

`DXPutLong(handle, Index, Value)`
Write a 4 byte signed integer to buffer

`DXPutSingle(handle, Index, Value)`
Write a 4 byte single precision IEEE Real to buffer

`DXPutDouble(handle, Index, Value)`
Write a 8 byte double precision IEEE Real to buffer

`DXPutString(handle, Index, Value, DataType)`
Write a zero delimited string to buffer

Which `DXPut` function you use primarily depends upon how the data is stored in your application. For example, if you are storing a numeric value in your application as a single precision IEEE Real, then you would write that data using the `DXPutSingle` function. In general, you should use the `DXPut` function which best reflects the way your data is stored in your application.

Note: that DataExport/DLL ***will not return an error*** *if it is required to fix-up a `DXPut` value.*

Typically, the `DXPut` function you use will also match the definition of the column to which you are writing, for instance, writing a Long Integer (`DXPutLong`) to a Long Integer column (column defined with a DataType of DXData_LONG). However, DataExport/DLL will *fix-up* inappropriate Put values to match the definition of a column. For example, the DLL will change a short integer into a long integer if it is being written to a long integer column. There are some obvious limitations to the fixing up the DLL can do, for instance writing a string “George” to a Boolean column—or any numeric column—will cause DataExport/DLL to write a zero value to the cell.

The following code shows some examples of DXPut commands. See Appendix A for complete function syntax and prototypes:

Visual Basic Example: `DXPut` commands

```
result = DX.DXPutString(hTable, 1, _⇔
            "12-01-95", DXString_Date_MDY)

result = DX.DXPutString(hTable, 2, _⇔
            "Atlanta", DXString_Text)
```

```
    result = DX.DXPutSingle(hTable, 3, 5000.0, 3)
```

In general, any `DXPut` function can be used to write to any defined column, however, this will not always produce desired results. The chart below shows the most appropriate `DXPut` function to use, based on the `DataType` definition of the column you are writing to:

Column/Field DataType Setting	Appropriate DXPut Command	Appropriate StringType
DXData_GENERAL	`DXPutString`	StringType:
		DXString_Default
DXData_BOOLEAN	`DXPutInt`	None
DXData_BYTE	`DXPutInt`	None
DXData_INTEGER	`DXPutInt`	None
DXData_LONG	`DXPutLong`	None
DXData_CURRENCY	`DXPutSingle`	None
	`DXPutDouble`	None
DXData_SINGLE	`DXPutSingle`	None
DXData_DOUBLE	`DXPutDouble`	None
DXData_DATE	`DXPutString`	StringTypes:
		DXString_Date_MDY
		DXString_Date_DMY
		DXString_Date_YMD
		DXString_Date_MY
		DXString_Date_YM
		DXString_Date_YD
DXData_TEXT	`DXPutString`	StringTypes:
		DXString_Text
		DXString_Lowercase
		DXString_Uppercase
		DXString_Caps
DXData_TIME	`DXPutString`	StringType:
		DXString_Time

See Appendix A for more information about the `DXPut` functions, DataTypes and StringTypes.

Committing a Row/Record to the Output File

After writing data to the DataExport/DLL buffer a row/record is completed by sending a `DXWriteBuffer` command to the DLL. This command completes the row/record and commits the current buffer to the output file.

```
DXWriteBuffer(handle)
```

A row/record is not automatically written to the output file after the last buffer cell is filled. Only the `DXWriteBuffer` function commits the current buffer information to a file. Sending this command does not ‘clear out’ the buffer; information in the buffer is simply copied to the output file. This architecture allows re-use of the current data in the creation of subsequent rows/records.

Closing the Output File

When you have finished writing data to an output file, the file must be completed using the `DXCloseFile` command. This function closes the output file and releases the DataExport/DLL buffer resources. After this command is issued, a new output file can be started using the `DXInitTrans` command.

```
DXCloseFile(handle)
```

Note: The component stays loaded until your application closes or you specifically unload the module.

Complete Cell-by-Cell Example

This section shows a complete routine for writing to an output file, not including querying of the DLL and user interface.

Visual Basic Example: Complete Cell-by-Cell output routine

```
' You must have added the file DXTRANS.VB to your
' project to have the declarations and constants
' available.

Option Explicit

Global MaxTypes&, FileOutputType&, TypeExt$(), ⇔
       TypeName$(), TypeofTrans$()


Sub DoCellTrans ()

    ' This routine initializes a new output file,
    ' defines three columns, writes two records and
    ' closes the file.


    Dim FileName$, FileType$, hTable&, result&


    ' FileName and FileType could easily be passed
    ' from the main program, as well as the data.

    FileName$ = "d:\di\demtext.dbf"

    FileType$ = "DBF4"


    ' start up the translation engine and get back
    ' the hTable

    result = DX.DXInitTransW(FileName, FileType, "", ⇔
              3, 1, hTable)
    ' define the columns/fields of the data

    result = DX.DXDefDataW(hTable, 1, DXData_Date, 8, ⇔
              0, "DATE")

    result = DX.DXDefDataW(hTable, 2, DXData_Text, ⇔
              12, 0, "CITY")

    result = DX.DXDefDataW(hTable, 3, DXData_General, ⇔
              12, 3, "UNITS")


    ' clear out the buffer

    result = DX.DXClearBuffer(hTable)


    ' put info into buffer for first record

    result = DX.DXPutStringW(hTable, 1, "12-01-95", ⇔
              DXString_Date_MDY)

    result = DX.DXPutStringW(hTable, 2, "Atlanta", ⇔
              DXString_Text)

    result = DX.DXPutSingle(hTable, 3, 5000.0, 3)


    ' write the buffer for the first record to disk

    result = DX.DXWriteBuffer(hTable)


    ' clear out the buffer

    result = DX.DXClearBuffer(hTable)


    ' put info into buffer for second record

    result = DX.DXPutStringW(hTable, 1, "2-14-95", ⇔
              DXString_Date_MDY)

    result = DX.DXPutStringW(hTable, 3, "New York", ⇔
              DXString_Text)

    result = DX.DXPutSingleW(hTable, 7, 500.0, 3)


    ' write buffer for the second record to disk

    result = DX.DXWriteBuffer(hTable)


    ' close the table handle

    Call DXCloseFile(hTable)

End Sub
```

Refer to Appendix A for complete function syntax and prototypes.

File-Based Translation

A File-Based Translation uses two disk files to create an output file: one for a data source and one to control the creation of the new file. The translation process also requires one method call to the DataExport/DLL to initiate and perform the translation. This method is most useful when your application already has a routine to export a CSV, fixed field file or print an ASCII report to disk.

Two files are required for a file-based translation: a raw data file which contains the data to be translated and a control file which describes the type and structure of the output file and, optionally, how to read the source file.

Note: For developers' using DataExport/DLL as a substitute for DataImport's translation capabilities, the raw data file can be any ASCII text report. In addition, you can use either a pre-defined DataImport Mask (MSK) file or an EDF to control translation of the report file. Except for these substitutions, you should use the DLL as described in this section.

Required Methods

There is only one method required, `DXLaunchEngine`, in a File-Based Translation with DataExport/DLL. This function is used to initiate the translation process and pass required parameters to the DLL. For more information, see "Running a Translation" below.

Raw Data File

The creation of a raw data file, or input file, is done in the code of your program, using any routines you choose. The file can also be provided by another application. The format of the file should be either a Comma Separated Variable (CSV) ASCII file or a fixed field ASCII file with carriage return/line feeds at the end of each record. Tab Separated Variable (TSV) files and ASCII files with other delimiters can also be used. See "INFILE Statement" section below for more information.

Comma Separated Variable (CSV) format

A CSV format file uses commas "," to separate cells/fields and encloses strings in quotation marks <">. Each record ends with a carriage return/line feed.

```
"400-234-242399","SMITH, RONALD","Y","MEDTECH ⇔
INDEMNITY","HMDSR88900-9980",.4903,"H. ⇔
NORMAND","HBST1","HBSTC3","Y"

"845-538-546839","DEAN, CHRISTINA","Y","ENSURE ⇔
MEDICAL","CYMD800-5480",.5431,"H. NORMAND",⇔
"RCVN04","RCTRZ2","Y"
```

Fixed Field ASCII format

In a fixed field file, each column/field is a fixed number of characters wide, regardless of the information contained in the cells of the column. Each record ends with a carriage return/line feed. This format should not be confused with a fixed *length* format, which is similar but lacks the carriage return/line feed character to delimit rows/records.

```
400-234-242399    SMITH, RONALD      YMEDTECH INDEMN
845-538-546839    DEAN, CHRISTINA    YENSURE MEDICAL
^                 ^                  ^^
```

Note: For developers' using DataExport/DLL as a substitute for DataImport's translation capabilities, the raw data file can be any ASCII text report.

Writing an Export Definition File (EDF)

The EDF is used to control the translation of a raw data file or report, including the name and type of output file to be written, the structure of the output file and how the data is formatted. This control file is required for any file-based translation with DataExport/DLL.

The EDF is a simple ASCII file that tells DataExport/DLL how to translate the raw ASCII data file, whether it is a CSV or fixed fielded text file. An EDF contains one statement (`STATEMENT=`) per line. The following statements are required for most EDF files:

`VERSION=`	specifies version of the EDF format used
`COLUMN=`	specifies each column definition, in sequence
`INFILE=`	specifies the input file
`OUTFILE=`	specifies the output file

At the minimum, the EDF contains its format version number (`VERSION=X.X`) and definitions of the columns/fields (`COLUMN=`) to be created in the output file. The EDF format is described in detail in Appendix B. This section of the manual explains the format of the typical EDF file which will translate either a CSV or a fixed field ASCII file.

Note: For developers' using DataExport/DLL as a substitute for DataImport's translation capabilities, you can use either a pre-defined DataImport Mask (MSK) file or an EDF for your translation process. Refer to the "Report Translation Statements" section below for more information about using the EDF format to translate a report file.

Version Statement

The EDF format requires a version statement at the beginning of the file to tell DataExport/DLL how to read the file. For the format described in this manual, the version number is 1.0. Be sure to review the README.TXT file for any changes to the EDF format and subsequent version number changes. Assuming you are using version 1.0 of the EDF format, your version statement should read:

```
VERSION=1.0
```

This statement *must* be the first line of your EDF in order for the DLL to read the file definitions. If this statement is placed anywhere else in the EDF, DataExport/DLL will not be able to read the file.

Column Definitions

The EDF must contain information about the structure of the output file to be written, specifically the definition of the columns to be output. If you are using a fixed field

ASCII file as you raw data file, the EDF must also contain information about how to read the input file.

If you are using a CSV file as your input file, your EDF file must contain COLUMN statements: one statement for each column/field that you want to create from your CSV file. The syntax for a column statement is shown below:

Syntax: `COLUMN=width[,startpos[,type[,dup ⇔ [,"name"]]]]`

The `width` parameter is the only required parameter for this statement however, there are some recommendations which you should consider, depending on the file format you are using for your input file.

CSV Input Files

With a CSV file, COLUMN statements are applied to the input file in the order that the column/fields appear. In other words, the first COLUMN statement is applied to the first field of all records in the CSV file, the second statement is applied to the second field, etc.

Each COLUMN statement should contain three items: a maximum character width for the output column (`width`), the type of data (`type`) and the name for the column/field (`"name"`).

```
COLUMN=width,,type,,"name"
```

The `width` parameter tells DataExport/DLL how wide to make the column/field in the output file you are defining. Proper use of the `width` parameter requires that you know the maximum width—in number of characters—of data items in the column.

The `type` of data indicates whether the data is numeric, text or another format. The type number corresponds to the StringType codes described in the “StringTypes” section of Appendix A.

The `name` parameter is not required for all output formats, but we recommend you provide a column/field name for all output formats. This measure will provide consistency and avoid output errors when the parameter is required.

Fixed Field Input Files

Each COLUMN statement for a fixed field input file should contain four items: a maximum character width for the output column (`width`), the starting position of a column in the input file (`startpos`), the type of data (`type`) and the name for the column/field (`"name"`).

```
COLUMN=width,startpos,type,,"name"
```

A fixed field file requires one more parameter than a CSV file for each COLUMN statement: the `startpos` parameter. This parameter specifies the beginning of a column in the input file.

With a fixed field file, the order of the COLUMN statements is the order that that the columns/fields will appear in the output file. However, since you can specify which column to read in the input file using the `startpos` parameter, the actual order of the columns in the output file need not be the same as that of the input file. For example, the first COLUMN statement in your EDF could specify a `startpos` of 40, which might be the beginning of the third column in your input file. In which case, the *third* column of you *input* file will be the *first* column of your *output* file. Thus, the `startpos` parameter and the order of the COLUMN statements allows you to re-arrange the order of you columns in the output file.

The `width` parameter tells DataExport/DLL how wide to make the column/field you are defining in the output file. Proper use of the `width` parameter requires that you know the maximum width—in number of characters—of data items in the column.

The `type` of data indicates whether the data is numeric, text or another format. The type number corresponds to the StringType codes described in the "StringTypes" section of Appendix A.

The `name` parameter is not required for all output formats, but we recommend you provide a column/field name for all output formats. This measure will provide consistency and avoid output errors when the parameter is required.

Input and Output file

In order for DataExport/DLL to perform a translation, it must know the name and location of the input file, and the name, location and output type of the output file. In a file-based translation, this information is provided one of two ways:

1. Using INFILE and OUTFILE statements in the EDF.
2. Using parameters in the `CmdLine` string when calling the `DXLaunchEngine` function.

We suggest always including the INFILE and OUTFILE parameters, since writing an EDF is required for a file based translation and because this option defines a 'default' input and output file. The input and output statements in the EDF can then be overridden with a `CmdLine` parameter in the `DXLaunchEngine` call.

INFILE Statement

The INFILE statement specifies the name and path of the input file, or raw data file, you are using for your file-based translation.

Syntax: `INFILE="string"[,fielddel#,stringdel#]`

The `string` parameter specifies the file name and path of the input file. This parameter is the only required item if you are using a fixed field input file.

If you are using a standard CSV file as your input file with a comma "," separating each cell/field and straight quotation marks <"> around text fields, then the string parameter is all that is required. However, if your input file uses non-standard delimiters or is a Tab Separated Variable (TSV) file, then you must specify the delimiters for cells/fields and the text string delimiters.

Both the field delimiter (`fielddel#`) and string delimiter (`stringdel#`) are specified as ASCII numbers (e.g., 9 for a Tab).

Examples:

For a typical tab delimited ASCII file named TAB-DELM.TXT the INFILE statement would be:

```
INFILE="c:\data\tab-delm.txt",9
```

For an ASCII file named SC-DELM.TXT and with semicolons ";" separating fields and apostrophes "`'`" delimiting text fields, the INFILE statement would be:

```
INFILE="c:\data\sc-delm.txt",59,39
```

OUTFILE Statement

The OUTFILE statement specifies the name, path and file format of the output file of your file-based translation.

Syntax: `OUTFILE="string","filetype"`

The `string` parameter specifies the file name and path of the output file. This parameter is required for a CSV or fixed field file translation.

The `filetype` parameter specifies the file format of the output file in a FileType text code. This parameter is required for a CSV or fixed field file translation. See "FileTypes" in Appendix A for more information.

Example EDF

The following is a sample EDF for translating a CSV file called TEXTFILE.CSV with four fields into a Access 2.0 file:

```
VERSION=1.0

INFILE="c:\data\textfile.csv"

COLUMN=6,,1,,"Cust#"

COLUMN=40,,1,,"Company"

COLUMN=20,,0,,"AmountOwed"

COLUMN=8,,3,,"Date"

OUTFILE="c:\data\textfile.mdb","MDB"

TABLENAME="Outstanding Invoices"
```

Notice that the first column—`Cust#`—is defined as a text field (1 = DXString_Text), this is to make sure any leading zeros in the customer numbers are retained.

The TABLENAME statement is not required, but can be included if you wish to provide a name for the table in Access. If no name is specified, DataExport/DLL provides a name of "Table1".

Special Function Statements

There are many more statements in the EDF which can be used to translate text files. However, unless you are using DataExport/DLL as a substitute for DataImport's translation capabilities, many of these features will not be of use to you. There are a few statements, however, that you may find useful:

`TABLENAME=`	specifies table name for Access output files
`SHEETNAME=`	specifies sheet name for Excel 5.0 output files
`CURRENCY=`	defines currency symbol (default = "$")
`THOUSAND=`	sets thousands symbol (default = ",")
`DECIMAL=`	sets decimal symbol (default = ".")
`CODEPAGE=`	sets ASCII code page (default = 437)
`CUSTOMDATE=`	sets interpretation of non-delimited dates
`CENTURY=`	sets interpretation of two-digit years
`MONTHS=`	sets month names (default is US month names)

More information about these statements is available in Appendix B.

Report Translation Statements

For developers using DataExport/DLL as a substitute for DataImport's translation capabilities, these statements allow you to use the special features of a Mask definition:

`TITLE=`	defines title lines in input file
`HEADING=`	defines column headings in input file
`INCLUDE=`	includes only specified lines in translation
`EXCLUDE=`	excludes specified lines from translation
`PAUSE=`	stops translation of lines
`RESUME=`	re-starts translation of lines
`REFPT=`	defines a Reference Point
`TAG=`	defines a Line Tag
`ADJUSTWIDTH=`	adds a space to output columns
`SKIPMODE=`	turns on/off line skip mode
`STARTCELL=`	defines first spreadsheet cell to write to
`SIGNEDOP=`	defines Signed Overpunch characters

For more information about these statements, refer to Appendix B in this manual and the *Users Guide and Reference Manual for DataImport*.

Running a Translation

After a raw data file and an EDF have been written to disk, the `DXLaunchEngine` function is used to run the translation. At the minimum, this call specifies the name and path where the EDF (or MSK) is located.

```
DXLaunchEngine (CmdLine)
```

The `CmdLine` parameter is essentially a way to pass command line-like switches to the DataExport/DLL. At the minimum, this parameter contains the file name and path of the EDF to be used in the translation.

```
CmdLine = (filter[,[input],[output],[type],
   [display],[confirm]] [/A] [/C] [/R] [/K] [/S[=f[,s]]])
```

If you followed the recommended EDF implementation—see "Writing an Export Definition File (EDF)" above—then you should only have to be concerned with these parameters:

`filter`	specifies the full name and path of the EDF (or MSK file)
`/A`	set translation to append data if there is an existing file (default is to overwrite an existing file)
`/R`	delete the EDF or MSK file upon completion
`/K`	delete the input file upon completion

See "File-Based Translation Methods" in Appendix B for more information.

After initiating the translation, no further interaction is required. The DLL will simply run the translation and return the appropriate error code indicating completion or an error if a problem was encountered.

Code Example: Running a File-Based Translation

The following example routines demonstrate the initiation of a DataExport/DLL translation using the `DXLaunchEngine` method.

Visual Basic Example: Running a file-based translation with the `DXLaunchEngine` method

```
' You must have added the file DXTRANS.VB to your
' project to have the declarations and constants
' available.

Sub DoFileTrans ()

    ' This routine performs a file based translation


    Dim InFileName$, OutFileName$, EdfFileName$, ⇔
        FileType$, CmdLine$, result&


    ' FileNames and FileType could easily be passed
    ' from the main program,

    InFileName$ = "c:\downloads\textfile.prn"

    EdfFileName$ = "c:\downloads\textfile.edf"

    OutFileName$ = "c:\downloads\dxtest.dbf"

    FileType$ = "DBF"


    CmdLine$ = EdfFileName$ & "," & InFileName$ & ⇔
               "," & OutFileName$ & "," & FileType$


    result = DX.DXLaunchEngine(CmdLine$)


End Sub
```

See Appendix A for the DLL command syntax and prototypes for C and Visual Basic.

Chapter 4: File Format Requirements and Limitations

The file formats supported by the DataExport/DLL have some requirements and limitations of which you should be aware. This chapter details what you should know about the supported file formats.

The basic message of this chapter, is this: spreadsheets and database formats *have finite limits* on the amount of data they can hold, therefore you *must* set some limits on how much data you output with DataExport/DLL.

There are five major categories of output formats supported by the DLL:

Spreadsheets	(Excel, Lotus 1-2-3)
Databases	(Access, dBase, Clarion)
Interchange	(CSV, DIF, SDF)
Text	(TXT, ASC)
Word Processing	(MS-Word, Word Perfect Mail Merge)

These format categories all have general requirements and limitations which are discussed in the sections of this chapter. A summary table is provided at the end of this chapter.

Setting Output Limits for Your Application

The limitations of the output formats supported by DataExport/DLL varies widely. Some formats—specifically older spreadsheets formats—will only hold a very limited set of data, while others—like CSV—will allow you to output data until your hard drive fills up.

For this reason, you should set some limits on the amount of data you allow your application (or user) to output, depending upon the file format to which you are writing.

Columns/Fields

The maximum number of columns/fields that DataExport/DLL will allow you to output is 255. This limitation is based on the fact that almost all the spreadsheet and database formats supported by the DLL have an actual limit of 255 columns/fields—except for the dBase II format, which is limited to 32 fields.

We suggest that you set a general output limit of 255 columns in your application to address this limitation. In addition, you should set up an error trapping function to catch error code 1005 when more than 32 columns are defined for a dBase II output file.

Rows/Records

DataExport/DLL sets no limits on the number of rows/records it can output. However, the DLL will return and error code if the maximum number of rows/records for the specified output format is exceeded.

We suggest that you use this error code as a conditional end to your output routine. You should also provide an error message in your application when this occurs, telling users that the maximum number of rows/records supported by the format has been reached.

Note: DataExport/DLL returns the TooManyRecords (1004) error when you attempt to write a row/record which exceeds the maximum number the format can hold. The offending row/record is *not* written to the file. At that point, you have the option to stop translation and close the file, as well as provide an error message to your user.

Spreadsheets

Spreadsheet programs generally have fewer input requirements and are more forgiving in terms of the data that is written to them. For example, putting a text string in a column of numbers is acceptable. However, spreadsheets typically have a somewhat restrictive limit to the number of rows/records per sheet they will support.

All spreadsheet formats supported by DataExport/DLL have a limit of 255 columns and most of them can hold a maximum of 8,192 rows of data. Some newer formats, like Excel, can hold up to 16,384 rows, while older formats, like Lotus WKS, can only support 2048 rows. With spreadsheet formats in general, you should be more concerned with the total number of rows you are outputting, and less concerned with the type and structure of data.

Excel

The Excel format supports a larger number of rows than most spreadsheet formats (65,536), however the Excel format did not incorporate multiple-sheets, or "workbooks", into its structure, until version 5.0. The multiple-sheet change in v5.0 also added a new requirement for Sheet Names, which is unique for spreadsheet formats.

Version	Row Limit	Multi-Sheet	Sheet Names	FileType	File Extensio n
v2.1	16,384	No	--	XLS	XLS
v3.0	16,384	No	--	XLS3	XLS

v4.0	16,384	No	--	XLS4	XLS
v5.0/7.0	16,384	**Yes**	**Yes**	XLS5	XLS
V97/2000/ XP	65,536	**Yes**	**Yes**	XLS8	XLS

Lotus 1-2-3

The 1-2-3 format supports 8,192 rows per sheet, except for version 1a, which only supports 2048 rows. Lotus 1-2-3 was one of the first formats to support multiple-sheet files, but does not incorporate sheet names.

Version	Row Limit	Multi-Sheet	Sheet Names	FileType	File Extension
v1A	2,048	No	--	WKS	WKS
v2.0	8,192	No	--	WK1	WK1
v3.0	8,192	Yes	No	WK3	WK3
v4.0, 5.0	8,192	Yes	No	WK4	WK4

Quattro, Quattro Pro

The Quattro format supports 8,192 rows per sheet. Multiple sheets per file are not supported until Quattro Pro version 5.0 for Windows.

Version	Row Limit	Multi-Sheet	Sheet Names	FileType	File Extension
Quattro	8,192	No	--	WKQ	WKQ
Quattro Pro	8,192	No	--	WQ1	WQ1
Quattro Pro 5.0 Windows	8,192	Yes	No	WB1	WB1

Symphony

The Symphony format supports 8,192 rows per sheet and does not support multiple sheets in any of its formats.

Version	Row Limit	Multi-Sheet	Sheet Names	FileType	File Extension
v1.0	8,192	No	--	WRK	WRK
v1.1, 1.2, 1.3, 2.x	8,192	No	--	WR1	WR1

Databases

Databases tend to be more restrictive in terms of the type of data that can be written to them. For example, writing a text string to a numeric column/field in a database is *not* acceptable and will generate errors. On the other hand, databases usually have a larger capacity than a spreadsheet.

The typical database format can hold around 4.3 billion records (2^{32}) with a maximum of 255 fields. One older format, dBase II, holds only 65,536 records, but it is the exception to the rule. In general, you should be more concerned with the type and structure of data you are outputting into a database, but less concerned with the total number of records.

Access

The Access database format is unique in its use of Tables which are essentially normal dBase-like databases within a larger structure. Tables require a name separate from the name of the output file. The Access format is also unusual in the way it measures its maximum capacity.

Access has a limit of 32,768 Tables assuming there are no Queries, Forms, Reports, Macros or Modules defined.

MDB 2.0 and earlier formats have a maximum file size of 1 Gigabyte. There are no limits on the number of records in a single table, however, a single record can contain a maximum of 2000 characters, not including Memo fields and OLE objects. There is also a limit of 255 fields within a Table and a limit of 32,768 Objects (Tables, Queries, Forms, Reports, Macros and Modules) within a single Access file.

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Exten-sion
v1.1	1 GB*	2000	255	MDB1	MDB
v2.0	1 GB*	2000	255	MDB	MDB
v3.0	1 GB	2000	255	MDB3	MDB
V4.0/2000	1 GB	2000	255	MDB4	MDB

*1 Gigabyte is the maximum size of a single Access table.

dBase (Approach, FoxPro, Clipper, Alpha, xBase)

If there is a standard database format for PCs, it is the dBase format. Many database management programs, including Approach, FoxPro, Clipper, Alpha and other xBase applications use dBase as their format.

dBase files have a standard limit of 4.3 billion records (2^{32} or 4,294,967,296), a maximum of 4000 characters per record and a limit of 255 fields/columns, except for the dBase II format (see below). A dBase file is equivalent to a Table in an Access file. Multiple dBase files can be associated with one another using a relational database manager.

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Extension
v2	**65,536**	**1000**	**32**	DBF2	DBF
v3	4.3 billion	4000	255	DBF3	DBF
v4	4.3 billion	4000	255	DBF4	DBF

Clarion

The Clarion format has a standard capacity for records and fields. The format has a maximum capacity of 4.3 billion records, a limit of 4000 characters per record and a maximum of 255 fields.

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Extension
Clarion	4.3 billion	4000	255	DAT	DAT

Interchange Formats

Interchange formats typically do not have any restrictions either on the structure or amount of data you put into them. However, interchange formats do tend to be a bit large, since they are typically in an ASCII format, rather than a compressed, binary format.

DIF Rowwise, Columnwise

DIF stands for Data Interchange Format and is commonly used on mainframe and minicomputers. A "Rowwise" DIF is organized by row, while a "Columnwise" DIF is organized by column. The Columnwise format is the most commonly used DIF.

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Extension
Columnwise	None	None	None[†]	CDIF	DIF
Rowwise	None	None	None[†]	RDIF	DIF

[†] DataExport/DLL's output is limited to 255 columns /fields.

SYLK Multiplan

The SYLK, or SYmbolic LinK, format originated in Microsoft's DOS-based Multiplan software (circa 1985). This format is not commonly used but may be supported by some older DOS programs.

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Extension
SYLK	None	None	None†	SLK	SLK

† DataExport/DLL's output is limited to 255 columns /fields.

XML 1.0

XML stands for Meta Language and is commonly used to transfer data on the internet.

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Extension
v1.0	None	None	None†	XML	XML

† DataExport/DLL's output is limited to 255 columns /fields.

Text Formats

Text formats, similar to interchange formats, typically do not have any restrictions either on the structure or amount of data you put into them.

Since text formats are ASCII based, they tend to be a bit large because they use more bits to store data than a binary format.

Comma Separated Values (CSV), ASCII Delimited

These two formats are identical, however, CSV is the more recognized of these formats. Both are provided for in DataExport/DLL. This format separates cells/fields using commas ",", has one record per line (ended by a carriage return/line feed) and surrounds text cells/fields with straight quotation marks <">. DataExport/DLL's implementation of the CSV format writes dates as a Lotus serial date (number of days since January 1, 1900) and time as a Lotus decimal day (midnight to midnight expressed as decimal between zero and one).

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Extension
CSV	None	None	None†	CSV	CSV
ASCII del.	None	None	None†	ASC	ASC

† DataExport/DLL's output is limited to 255 columns /fields.

Tab Separated Values (TSV)

The TSV format is very similar to the CSV format, except that the cells/fields are separated by Tab characters instead of commas. The TSV format does not have quotation marks around text cells/fields. Dates are output as a Lotus serial date and times as a Lotus decimal day.

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Extension
TSV	None	None	None[†]	TSV	TSV

[†] DataExport/DLL's output is limited to 255 columns /fields.

Custom Delimited (UDD)

The Custom delimited format or User-Defined Delimited format allows you to specify the field and string delimiters in an ASCII delimited file. Dates are output as a Lotus serial date and times as a decimal day.

This output format is only available using a field-based translation method. Delimiters are specified using the UDD statement in an EDF.

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Extension
Custom del.	None	None	None[†]	UDD	UDD

[†] DataExport/DLL's output is limited to 255 columns /fields.

Fixed Field ASCII

The fixed length file format is a mainframe format in which each column/field is a specific number of characters wide and each record is a fixed number of characters long. This format contains no carriage return/line feeds.

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Extension
Fixed Len.	None	None	None[†]	FXD	FXD

[†] DataExport/DLL's output is limited to 255 columns /fields.

Standard Data Format (SDF)

The Standard Data Format, or SDF, is very similar to the fixed length format in that each column/field is a fixed number of characters wide. However, a SDF file ends each record with carriage return/line feed.

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Exten-sion
SDF	None	None	None†	SDF	SDF

† DataExport/DLL's output is limited to 255 columns /fields.

Print Image (PRN), Word Processing Text

The Print Image and Word Processing text formats are simply the output of data with all included spaces. These formats will typically look like a SDF file, however these file formats accept title and header information.

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Exten-sion
Print Im.	None	None	None†	PRN	PRN
Word P.T.	None	None	None†	TXT	TXT

† DataExport/DLL's output is limited to 255 columns /fields.

HTML Table

The HTML Table formats is a simple tabular representation of data that can be viewed in a web browser.

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Exten-sion
V3.0	None	None	None†	HTM	HTM

† DataExport/DLL's output is limited to 255 columns /fields.

Named Values

The Named Value output type formats is simply the fieldname/column name followed by the value.

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Exten-sion
Named Val	None	None	None†	NVL	NVL

† DataExport/DLL's output is limited to 255 columns /fields.

Word Processing Formats

Word processing formats supported by DataExport/DLL are data files used for mail merge routines and should not be confused with formatted documents like DOC or WRI files. These files contain only data such as names and addresses.

As with CSV and other interchange formats, Word Processing merge files have very few limits either on the structure or the amount of data they can contain. However, unlike CSV files, Word Processing formats *do* require column/field names for output.

Microsoft Word Mail Merge

The Microsoft Word Mail Merge format used for creating merged documents, such as multiple copies of a letter with different mailing addresses. This format is very similar to a CSV file, however, the first row/record of this file provides the names of the fields in the file, so column/field names are required for this output format.

Version	Record Limit	Record Width Limit	Field Limit	FileTyp e	File Exten-sion
MS Merge	None	None	None†	WRD	WRD

† DataExport/DLL's output is limited to 255 columns /fields.

WordPerfect Mail Merge

The Word Perfect Mail Merge format used for creating merged documents, such as multiple copies of a letter with different mailing addresses. This format also uses names for columns/fields and should be provided for in your output definitions.

Version	Record Limit	Record Width Limit	Field Limit	FileType	File Exten-sion
v5.0.	None	None	None†	W50	W50
v5.1	None	None	None†	W51	W51

† DataExport/DLL's output is limited to 255 columns /fields.

Summary Table

Version	Row/ Record Limit	Record Width Limit	Column/ Field Limit	FileType	File Exten-sion
Access 1.1	1 GB*	2000	255	MDB1	MDB
Access 2.0	1 GB*	2000	255	MDB	MDB
ASCII del.	None	None	None†	ASC	ASC
Clarion	4.3 billion	4000	255	DAT	DAT
CSV	None	None	None†	CSV	CSV
dBase v2	65,536	1000	32	DBF2	DBF
dBase v3	4.3 billion	4000	255	DBF3	DBF
dBase v4	4.3 billion	4000	255	DBF4	DBF
DIF Column	None	None	None†	CDIF	DIF
DIF Rowwise	None	None	None†	RDIF	DIF
Excel v2.1	16,384	None	255	XLS	XLS
Excel v3.0	16,384	None	255	XLS3	XLS
Excel v4.0	16,384	None	255	XLS4	XLS
Excel v5.0	16,384	None	255	XLS5	XLS
Excel 97/2000/XP	65,536	None	255	XLS8	XLS
Fixed Length	None	None	None†	FXD	FXD
Lotus 1-2-3, 4.0, 5.0	8,192	None	255	WK4	WK4
Lotus 1-2-3 1A	2,048	None	255	WKS	WKS
Lotus 1-2-3 2.0	8,192	None	255	WK1	WK1
Lotus 1-2-3 3.0	8,192	None	255	WK3	WK3
MS Word	None	None	None†	WRD	WRD

Version	Row/ Record Limit	Record Width Limit	Column/ Field Limit	FileType	File Exten-sion
Print Image	None	None	None†	PRN	PRN
Quattro	8,192	None	255	WKQ	WKQ
Quattro Pro	8,192	None	255	WQ1	WQ1
Quattro Pro 5.0 Windows	8,192	None	255	WB1	WB1
Standard Data Format (SDF)	None	None	None†	SDF	SDF
SYLK	None	None	None†	SLK	SLK
Symphony v1.0	8,192	None	255	WRK	WRK
Symphony v1.1, 1.2, 1.3, 2.x	8,192	None	255	WR1	WR1
TSV	None	None	None†	TSV	TSV
Word Proc.	None	None	None†	TXT	TXT
WordPerf. 5.0	None	None	None†	W50	W50
WordPerf. 5.1	None	None	None†	W51	W51

*1 Gigabyte is the maximum size of a single Access table.

† DataExport/DLL's output is limited to 255 columns /fields.

Appendix A: DataExport/DLL Methods

This chapter is a reference guide to the DataExport/DLL Interface. The guide provides a listing of each method general use and syntax.

The syntax and parameter definitions are provided using standard C programming conventions for variables:

INT	2 byte signed short integer
DOUBLE	8 byte floating point IEEE real
FLOAT	4 byte floating point IEEE real
LONG	4 byte signed long integer
LPLONG	long pointer to a long integer
LPSTR	long pointer to a string buffer. All strings use the ASCIIZ (ASCII zero delimited) C standard type.

Visual Basic programmers should not be as concerned whether the parameter is passed as a normal variable or as a long pointer, this is handled by using the function declarations provided in DXTRANS.VB. Just add this file to your Visual Basic project.

Methods for Querying Supported Formats

DataExport/DLL provides an interface for determining the file formats currently supported by the DLL. This interface can be used by you to auto-generate dialog boxes and routines at run time which automatically reflect new output formats supported by DataExport/DLL.

DXGetNumTypes

Syntax: LONG `DXGetNumTypes()`

Description: Returns the number of output types supported by the current version of DXTRANS.DLL installed on the local machine.

Return Value: Returns the number of supported output types (LONG) or a 0 if unsuccessful.

DXEnumTypes

Syntax: `LONG DXEnumTypes(Index, TypeDesc, TypeExt, TypeID, &TypeofTrans, &NameNeeded)`

Parameter	Type	Description
`Index`	LONG	Output type # (1 based) obtained from `DXGetNumTypes()`
`TypeDesc`	LPSTR	Pointer to buffer to receive Menu name of output type. This buffer must be allocated to hold up to 40 characters.
`TypeExt`	LPSTR	Pointer to buffer to receive file extension of output type. This buffer must be allocated to hold up to 4 characters.
`TypeID`	LPSTR	Pointer to buffer to receive FileType used in `DXInitTrans` of output type (see "FileType" section). This buffer must be allocated to hold up to 5 characters.
`TypeofTrans`	LPLONG	Pointer to integer to receive Category of output type: 0 = text 1 = mail merge 2 = interchange 3 = spreadsheet 4 = database
`NameNeeded`	LPLONG	Pointer to integer to receive `NameNeeded` flag; returns -1 if output type requires a `TableName` in `DXInitTrans`

Description: Provides information about an individual format—specified by an output type Index number (1 through X, where X equals the value returned by `DXGetNumTypes`).

Return Value: Success/Error code. See "Return Codes" section for definition.

Cell-by-Cell Output Methods

These methods are the primary interface with DataExport/DLL. It provides controls for sending data from your program into an output file.

DXInitTrans

Syntax: `LONG DXInitTrans(FileName, FileType, TableName, NumCol, IfExistFlag, &handle)`

Parameter	Type	Description
`FileName`	LPSTR	Pointer to buffer that contains Name of file to be output
`FileType`	LPSTR	Pointer to buffer that contains FileType (see “FileType” section)
`TableName`	LPSTR	Pointer to buffer that contains Name of Table (64 characters maximum)
`NumCol`	LONG	Number Columns/Fields to be output (255 maximum)
`IfExistFlag`	LONG	Flag that specifies whether to overwrite or append if the specified `FileName` exists: 0 = Overwrite 1 = Append
`handle`	LPLONG	Pointer to integer to receive handle number, or 0 if error

Description: The `DXInitTrans` function initializes an output file, specifying the name and type of file, the number of columns/field in the file and determines how DataExport/DLL will treat an existing file with a file name matching the specified `FileName`.

Return Value: Success/Error code. See "Return Codes" section for definition.

Related Functions: `DXDefData, DXCloseFile`

DXDefData

Syntax: `LONG DXDefData(handle, Index, DataType, DataWidth, DataDecimal, DataName)`

Parameter	Type	Description
`handle`	LONG	Handle number returned from `DXInitTrans`
`Index`	LONG	Column/field number (1 based), 1 through number of columns specified by `NumCol` in the preceding `DXInitTrans` call
`DataType`	LONG	DataType (1-10) see “DataTypes” section
`DataWidth`	LONG	Width of column/field in characters
`DataDecimal`	LONG	Number of decimals in column/field data
`DataName`	LPSTR	Pointer to buffer that contains the name of the column/field

Description: The `DXDefData` function specifies the width and type of data in each column/field. This function must be called once for each of the columns/fields specified by `NumCol` in the preceding `DXInitTrans`.

Data in the buffer is kept until overwritten or cleared out with `DXClearBuffer`

Once all column/fields have been defined, DataExport/DLL initializes a buffer into which the calling application can write the data for each row/record using the `DXPut` functions. Each row/record is committed to the output file with the `DXWriteBuffer` function.

Return Value: Success/Error code. See "Return Codes" section for definition.

Related Functions: `DXInitTrans`

DXPutInt

Syntax: `LONG DXPutInt(handle, Index, Value)`

Parameter	Type	Description
`Handle`	LONG	Handle number returned from `DXInitTrans`
`Index`	LONG	Column/field number (1 based)
`Value`	int	2 byte signed Integer

Description: The `DXPutInt` function writes a two byte, signed integer value to the specified column/field index in the initialized buffer.

Return Value: Success/Error code. See "Return Codes" section for definition.

Related Functions: `DXPutLong, DXPutSingle, DXPutDouble, DXPutString`

DXPutLong

Syntax: `LONG DXPutLong(handle, Index, Value)`

Parameter	Type	Description
`handle`	LONG	Handle number returned from `DXInitTrans`
`Index`	LONG	Column/field number (1 based)
`Value`	LONG	4 byte signed Long Integer

Description: The `DXPutLong` function writes a four byte, long integer value to the specified column/field index in the initialized buffer.

Return Value: Success/Error code. See "Return Codes" section for definition.

Related Functions: `DXPutInt, DXPutSingle, DXPutDouble, DXPutString`

DXPutSingle

Syntax: `LONG DXPutSingle(handle, Index, Value)`

Parameter	Type	Description
`Handle`	LONG	Handle number returned from `DXInitTrans`
`Index`	LONG	Column/field number (1 based)
`Value`	float	4 byte Single Precision IEEE Real

Description: The `DXPutSingle` function writes a four byte, single precision IEEE Real value to the specified column/field Index in the initialized buffer.

Return Value: Success/Error code. See "Return Codes" section for definition.

Related Functions: `DXPutInt, DXPutLong, DXPutDouble, DXPutString`

DXPutDouble

Syntax: `LONG DXPutDouble(handle, Index, Value)`

Parameter	Type	Description
`handle`	LONG	Handle number returned from `DXInitTrans`
`Index`	LONG	Column/field number (1 based)
`Value`	double	8 byte Double Precision IEEE Real

Description: The `DXPutDouble` function writes an eight byte, double precision IEEE real value to the specified column/field Index in the initialized buffer.

Return Value: Success/Error code. See "Return Codes" section for definition.

Related Functions: `DXPutInt, DXPutLong, DXPutSingle, DXPutString`

DXPutString

Syntax: `LONG DXPutString(handle, Index, Value, DataType)`

Parameter	Type	Description
`handle`	LONG	Handle number returned from `DXInitTrans`
`Index`	LONG	Column/field number (1 based)
`Value`	LPSTR	Pointer to zero delimited String
`DataType`	LONG	StringType (0-14)—see “StringTypes” section in this chapter

Description: The `DXPutString` function is used to write text string values into the specified column/field Index in the initialized buffer. The function can also be used to parse certain string values into a more appropriate format. For example, putting the string “12/31/95” and defining it as a date (“DXString_Date_MDY” StringType) will cause DataExport/DLL to write the data into the output file in the appropriate date format.

Return Value: Success/Error code. See "Return Codes" section for definition.

Related Functions: `DXPutInt, DXPutLong, DXPutSingle, DXPutDouble`

DXWriteBuffer

Syntax: `LONG DXWriteBuffer(handle)`

Parameter	Type	Description
handle	LONG	handle number returned from `DXInitTrans`

Description: The `DXWriteBuffer` function commits the data currently in the buffer to a row/record in the output file. *Note:* Data in the buffer is not automatically cleared out by this command; data remains until overwritten or cleared out with the `DXClearBuffer` function.

Return Value: Success/Error code. See "Return Codes" section for definition.

Related Functions: `DXClearBuffer`

DXClearBuffer

Syntax: `LONG DXClearBuffer(handle)`

Parameter	Type	Description
handle	LONG	handle number returned from `DXInitTrans`

Description: The `DXClearBuffer` function erases any data in the currently initialized DataExport/DLL buffer.

Return Value: Success/Error code. See "Return Codes" section for definition.

Related Functions: `DXWriteBuffer`

DXCloseFile

Syntax: `void DXCloseFile(handle)`

Parameter	Type	Description
handle	LONG	handle number returned from `DXInitTrans`

Description: The `DXCloseFile` function closes the current output file. The expected last function call is a `DXWriteBuffer`.

Return Value: Does not return value.

Related Functions: `DXInitTrans`

File-Based Translation Method

This method is used to initiate a translation in the file based translation method. This single function operates similarly to an executable call with command-line parameters.

DXLaunchEngine

Syntax: `LONG DXLaunchEngine(CmdLine)`

Parameter	Type	Description
`CmdLine`	LPSTR	Pointer to buffer that contains the command line text, see “Command Line Parameters” below

Description: The `DXLaunchEngine` function initiates a file-based translation using the DataExport/DLL. An Export Definition File (EDF) or a DataImport Mask (MSK) file must be present to define the data structure of the output file and translation method. A Raw Data File—or ASCII text report file—must also be present as a data source.

Return Value: Success/Error code. See "Return Codes" section for definition.

Related Functions: None

Command Line Parameters

Parameters for the `DXLaunchEngine` function are similar to command line switches used when running an executable (EXE) file.

Note: For developers familiar to the DataImport command line, the switches and parameters for `DXLaunchEngine` are nearly identical to that of DIW.EXE command line. If your are familiar with the DataImport command line parameters, you should move on to the next section “Differences from the DataImport Command Line” Otherwise, please continue with this section.

Once an EDF or MSK file has been defined and saved to disk, the translation can be initialized using these command line controls:

Syntax:

```
CmdLine = "filter[,[input],[output],[type],
   [display],[confirm]] [/A] [/C] [/R] [/K] [/S[=f[,s]]]"
```

The command line parameters for the `DXLaunchEngine` function are positional and separated by commas. If a parameter is skipped, a comma must be used to hold its place. The switches, `/A`, `/C`, `/R`, `/K` and `/S` are not positional and are not separated by commas.

The only required parameter is the `filter` (an EDF or Mask File name). If no other parameters are provided, the instructions specified in the EDF or MSK will be used in the translation.

`filter` EDF or Mask File name, including the path if necessary. *This is the only required parameter.* If no other parameters are provided, the instructions specified in the EDF or MSK will be used in the translation.

Note: The following command-line switches take precedence over any definitions in the provided EDF or MSK file. For example, if an EDF specifies a file name of "outfile1.xls" and the command line specifies a file name of "file5.xls", the filename "file5.xls" will be used.

`input` Input File name, including the path and extension.

`output` Output File name, including the path if necessary. If an extension is specified, it will be used rather than the file extension DataImport normally uses based on the type of translation.

`type` Type of translation to be performed. See FileTypes section of this chapter for type strings.

`display` Specifies whether the output is to be displayed on screen during translation: **Y** for yes, **N** for no. The default is Yes.

`confirm` Specifies whether Include Line and Exclude Line treatments must be confirmed manually during the translation. **Y** for yes, **N** for no. The default is No.

`/A` Appends the output of the translation to the end of an existing Output File.

`/C` Combines the output of the translation into an existing spreadsheet Output File.

`/R` Removes the Mask or EDF file after reading information from the file. Without this switch, the `filter` file remains on disk.

`/K` Kills the input file after the translation is complete. Without this switch, the input file is left on disk.

`/S[=f[,s]]` Specifies the delimiters for a custom delimited input file; where f is the ASCII number of the field separator (44 for comma), and s is the ASCII number of the string delimiter (34 for straight quote). Comma and quotation mark is the default if not specified.

To specify some parameters and not others, include the intervening commas as place holders. This is necessary to indicate which of the options you want to use. For example, if you want to use the name of the Input File stored with the Mask, but want to change the name of the Output File, you would place two commas before the name of the new Output File. Otherwise, DataExport/DLL would interpret the output file name as an Input File. Commas, however, are unnecessary as place holders before the "/" switches, such as the /A and /C parameters.

If a parameter is not specified on the command line and the parameter has not been specified in the EDF or MSK, the translation cannot proceed. In such cases, the translation is aborted and a message is displayed on the screen to indicate the missing or invalid parameter(s).

The following four examples illustrate the ways translations can be initiated using a command line.

Translate Command Line Example 1

To perform a file-based translation using the following parameters:

EDF name	FILTER.EDF
Input File name	DIDEMO.TXT

Output File name	SALESDAT
Translation type	XLS
Display on	Y (Yes)
Confirm include/exclude	Y (Yes)
Append to existing file	/A

The `CmdLine` parameter should be:

```
"FILTER.EDF,DIDEMO.TXT,SALESDAT,XLS,Y,Y /A"
```

Translate Command Line Example 2

To perform a file-based translation using the following parameters:

Mask File name	MYMASK.MSK
Input File name	as specified in the mask
Output File name	as specified in the mask
Translation type	XLS
Display on	default to yes
Confirm include/exclude	default to no

The `CmdLine` parameter should be:

```
"MYMASK.MSK,,,XLS"
```

Translate Command Line Example 3

To perform a file-based translation using the following parameters:

EDF name	COMPLET.EDF
Input File name	as specified in the EDF
Output File name	as specified in the EDF
Translation type	as specified in the EDF
Display on	default to yes
Confirm include/exclude	default to no

The `CmdLine` parameter should be:

```
"COMPLET.EDF"
```

Translate Command Line Example 4

To perform a file-based translation using the following parameters:

Mask File name	MYMASK.MSK
Input File name	ORIGINAL.DAT
Output File name	GOOD
Translation type	as specified in the mask
Display on	default to yes
Confirm include/exclude	default to no

File-combine	/C

The `CmdLine` parameter should be:

```
"MYMASK.MSK,ORIGINAL.DAT,GOOD /C"
```

Differences from the DataImport Command Line

There are three additional switches for the command line parameters of the `DXLaunchEngine` function. Two of these switches concern the treatment of EDF, MSK and data files after completing translation. The third allows you to specify delimiters for a custom-delimited ASCII input file.

`/R` Removes the Mask or EDF file after reading information from the file. Without this switch, the `filter` file remains on disk.

`/K` Kills the input file after the translation is complete. Without this switch, the input file is left on disk.

`/S[=f[,s]]` Specifies the delimiters for a custom delimited input file; where f is the ASCII number of the field separator (44 for comma), and s is the ASCII number of the string delimiter (34 for quote). Comma and quote is the default.

FileTypes

These codes are used in the `FileType` parameter of the `DXInitTrans` command to define the type of output format to be written. `FileType` codes for each format can be obtained at run time using the method `DXEnumTypes`. For more information about these file formats, refer to Chapter 4: File Format Requirements and Limitations.

ASCII text file	**TXT**
Borland Quattro	**WKQ**
Borland Quattro Pro	**WQ1**
Borland Quattro Pro 5.0 for Windows	**WB1**
Clarion	**DAT**
Columnwise DIF	**CDIF**
Columnwise DIF	**DIF**
Comma Separated Variable	**CSV**
Comma separated ASCII	**ASC**
dBase II	**DBF2**
dBase III	**DBF3**
dBase IV	**DBF4**
Fixed record format without delimiters	**FXD**
HTML Table	**HTM**
Lotus 1-2-3 release 1 and 1A	**WKS**
Lotus 1-2-3 release 2.x	**WK1**

Lotus 1-2-3 release 3.x	**WK3**
Lotus 1-2-3 release 4.x and 5.x	**WK4**
Microsoft Access 1.1	**MDB1**
Microsoft Access 2.0	**MDB**
Microsoft Access 3.0	**MDB3**
Microsoft Access 4.0/2000/XP	**MDB4**
Microsoft Excel version 2.1	**XLS**
Microsoft Excel version 3.0	**XLS3**
Microsoft Excel version 4.0	**XLS4**
Microsoft Excel version 5.0	**XLS5**
Microsoft Excel version 8.0/97/2000/XP	**XLS8**
Microsoft Word data document	**WRD**
Named Values	**NVL**
Print Image	**PRN**
Rowwise DIF *	**RDIF**
Standard Data Format	**SDF**
SYLK or Symbolic Link	**SLK**
Symphony release 1.0	**WRK**
Symphony release 1.1, 1.2 and 2.x	**WR1**
Tab separated variables	**TSV**
User-defined delimited	**UDD**
Word Perfect 5.0 secondary merge file	**W50**
Word Perfect 5.1 secondary merge file	**W51**
XML 1.0	**XML**

* Only available in a File-Based Translation

DataTypes

These codes are used in the `DataType` parameter of the `DXDefData` method to define the type of data in a Column or Field.

Variable Name	Value	Data Type Description
DXData_GENERAL	0	Allows DataExport/DLL to define data type based on data item passed *
DXData_BOOLEAN	1	Numeric value of 1 (Yes) or 0 (No)
DXData_BYTE	2	1 byte integer
DXData_INTEGER	3	2 byte signed integer
DXData_LONG	4	4 byte signed integer
DXData_CURRENCY	5	Numeric currency value
DXData_SINGLE	6	4 byte single precision IEEE Real
DXData_DOUBLE	7	8 byte double precision IEEE Real
DXData_DATE	8	Date formatted value
DXData_TEXT	9	Text string to be written as-is or formatted, based on passed StringType parameter
DXData_TIME	10	Time formatted value

* DataExport/DLL will attempt to identify the first data item as a numeric value and, failing that, will assume it is a text item. This decision determines the data type setting of column/field. The DXData_GENERAL setting is typically only used with spreadsheets or for writing text items to an output file.

These values are declared as constants in the DXTRANS.H and DXTRANS.VB declaration files included with DataExport/DLL.

Note: Not all output formats support each of these DataTypes directly. When an data type is not directly supported by the output format, DataExport/DLL selects the closest data type the output format will support. In general, you should be as specific as possible about the type of data you want to output when defining columns/fields and `DXPut`-ing data to the buffer. The DLL will then determine the most appropriate final data format for the output file.

For more information on the relationship between DataTypes, `DXPut` functions and StringTypes, refer to the table in "Writing Cells in a Row/Record" in Chapter 3.

StringTypes

These codes are used in the `DataType` parameter of the `DXPutString` method to define the how the passed text string should be formatted.

Variable Name	Value	Formatting Description
DXString_Default	0	DataExport/DLL will first try to output the string as a number and then fallback to outputting the string as text
DXString_Text	1	DataExport/DLL writes the string as text, into the output file as-is
DXString_SOP	2	Translates string in a signed overpunch format into a numeric value
DXString_Date_MDY	3	Formats a delimited string* as a date of the format Month, Day, Year
DXString_Date_DMY	4	Formats a delimited string* as a date of the format Day, Month, Year
DXString_Date_YMD	5	Formats a delimited string* as a date of the format Year, Month, Day
DXString_Date_MY	6	Formats a delimited string* as a date of the format Month, Year
DXString_Date_YM	7	Formats a delimited string* as a date of the format Year, Month
DXString_Date_YD	8	Formats a delimited string* as a date of the format Year, Day,
DXString_Date_DY	9	Formats a delimited string* as a date of the format Day, Year
DXString_Date_Custom	10	Formats a non-delimited string as a date **
DXString_Time	11	Formats a string as a time†
DXString_Lowercase	12	Formats text string to all lowercase characters
DXString_Uppercase	13	Formats text string to all UPPERCASE characters
DXString_Caps	14	Formats text string so that the initial letter of each word is capitalized

* Delimited date strings are numeric and can be delimited with any non-numeric character, such as “/”, “-”, “\”, etc.

** This feature is only available in a File-Based translation. The custom, non-delimited format must be specified in the EDF (using CUSTOMDATE) or pre-defined in the MSK file.

† Strings to be formatted as a time must be in one of two forms: XX:XX[AM/PM] or as XX:XX in 24-hour format.

These values are declared as constants in the DXTRANS.H and DXTRANS.VB declaration files included with DataExport/DLL.

For more information on the relationship between DataTypes, `DXPut` functions and StringTypes, refer to the table in “Writing Cells in a Row/Record” in Chapter 3.

CellTypes

DXSetFormat method to define the how the columns of cells should be formatted in a spreadsheet.

Variable Name	Value	Formatting Description
DXNone	0	DataExport/DLL will not format the cell with any specific commas or currency symbols in a spreadsheet.
DXCellCurrency	16	DataExport/DLL will format the cell with currency symbols.
DXCellComma	32	DataExport/DLL will format the cell with commas.

UDD Strings

These codes are used in the `DXDefStringA` and `DXDefStringA` methods to define which strings should be used for the field separator and the string delimiter in a UDD translation.

Variable Name	Value	Formatting Description
DXStrInd_FieldSeparator	1	This indicates that the string passed should be used for the Field Separator.
DXStrInd_StringDelimiter	2	This indicates that the string passed should be used for the String Delimiter.

Return Codes

DataExport/DLL methods can return a variety of error codes generated internally, by the operating system or other DLLs.

These are the error codes returned specifically by DataExport/DLL:

Error Mnemonic	Code
DXErr_OK	-1
DXErr_NoHandlesAvail	1001
DXErr_HandleNotValid	1002
DXErr_IndexNotValid	1003
DXErr_TooManyRecords	1004

DXErr_BadDataType	1005
DXErr_BadStringType	1006
DXErr_TooManyColumns	1007
DXErr_BadFieldName	1008

1001 No handles are available. Typically this error results when you have previously called DataExport/DLL and it has not completed it's translation session. In this version of DataExport/DLL, only one handle is available for translation at one time.

1002 Handle provided is not valid. The handle you provided with your call is not the one returned to your program with result of the `DXInitTrans` function.

1003 Index provided is not valid. The column/field Index provided with the last call does not fall within the range of columns/fields you set in the your last `DXInitTrans` call (e.g., You are trying to write to column 10 when you only have 9 columns defined.)

1004 Too many rows/records have been output. Some formats—specifically spreadsheet formats—have a limit to the number of rows they can hold per sheet, see Chapter 4: File Format Requirements and Limitations for more information. Database formats are typically limited to 4.3 billion records (2^{32}), although dBase 2 is limited to 65,536 records. Text, mail merge and interchange formats do not have limitations on the number of rows/records they can hold.

1005 Bad `DataType` code received. DataExport/DLL has received a `DataType` number it does not recognize in a `DXDefData` function call.

1006 Bad `StringType` code received. The DLL has received a `StringType` number it does not recognize in a method.

1007 Too many columns/fields have been defined. The vast majority of spreadsheet and database formats supported by DataExport/DLL have a limit of 255 columns/fields per file (or per Access table), except for dBase II, which has a limit of 32 fields. The DLL can output a maximum of 255 fields, see Chapter 4: File Format Requirements and Limitations for more information.

These values are declared as constants in the DXTRANS.H and DXTRANS.VB files included with DataExport/DLL. Additional common error codes *not* generated by DataExport/DLL are also provided in these header files. These codes are generated by the operating system and other DLLs.

1008 The field name passed with DXDefData is not valid for the output type passed.

Appendix B: EDF Statements and Format

An Export Definition File (EDF) provides the controls to conduct a file-based translation from an ASCII Raw Data File to a new output file in the format you specify. This chapter describes the control functions and the format of the EDF file.

EDF Format

This section discusses the EDF format, its requirements and arrangement of statements. The EDF format is a simple ASCII text file and has only a few requirements for definition.

An EDF is made up of statements (e.g., STATEMENT=X,Y,"Z"), each statement is written with its parameters on a line by itself. The statement text to the left of the equal sign "=" is not case sensitive, hence

```
STATEMENT=

statement=

Statement=
```

are all acceptable syntax. Values to the right of the equal sign, however *are* case sensitive. Any text strings in the parameters of a statement should be surrounded by straight quotation marks <">. Blank lines are allowed and comment lines can be inserted with a semicolon ";" at the beginning of a line.

Only the `VERSION`, `COLUMN` (or `REFPT` and `TAG)`, `INFILE` and `OUTFILE`, statements are required in an EDF. The "Required Statements" and "Statement Order" sections below explain the use of and special considerations for these statements.

Required Statements

There are only two required statements in an EDF file and two more statement/definitions which must be provided either in the EDF or the `CmdLine` parameter of the `DXLaunchEngine` API call. The following two statements are required in any EDF:

`VERSION` is required to identify the format of the EDF file.

`COLUMN` is required to identify the number, size and type of the columns/fields to be written to the output file. One COLUMN statement is required for each column/field to be output. (If you are using REFPTs and TAGs, no column statements are required, but at least one REFPTs and one TAG must be defined.)

The information provided by the following two statements is required to be provided either in the EDF or the `CmdLine` parameter of the `DXLaunchEngine` API call:

`INFILE` identifies the data source file (either delimited ASCII or a text report) which DataExport/DLL is to translate. This statement is equivalent to the `input` parameter of the `CmdLine` in the `DXLaunchEngine` function.

`OUTFILE` identifies the output file which DataExport/DLL is to write. This statement is equivalent to the `output` parameter of the `CmdLine` in the `DXLaunchEngine` function.

If these parameters are provided in both the EDF and the `CmdLine` parameter, the `CmdLine` definitions will be used.

Statement Order

There are only three statements in the EDF format which are required to be in a specific order in the file:

`VERSION` must be the first statement of any EDF file. The placement of this statement is crucial for proper interpretation of an EDF.

`REFPT` statements can be located on any line of the EDF file except for the first line. However, its related `TAG` statements must follow directly after it.

`TAG` statements *must* follow the REFPT statement on which they are based.

All other statements can be placed anywhere in the EDF, except on the first line of the file.

Examples

This section provides a few examples of properly constructed EDF files.

Example 1

The following EDF example translates a CSV input file into an Access format file:

```
VERSION=1.0

INFILE="c:\data\contacts.csv",44,34

COLUMN=20,0,,,"FirstName"

COLUMN=40,0,1,,"LastName"

COLUMN=40,0,1,,"Company"

COLUMN=8,0,5,,"DateLastContacted"

OUTFILE="c:\data\prospect.mdb",MDB

TABLENAME="Contacts"
```

Notice that the `startpos` is set to zero for each of the COLUMN statements, since the source file is delimited. The columns are defined in the order in which they appear in the CSV file; the first COLUMN statement defines the first column/field in the CSV file, the second statement defines the second column, etc. The optional TABLENAMEstatement is used only for Access formats.

Example 2

The following EDF example translates a fixed fielded input file into an Lotus 1-2-3 version 4 spreadsheet:

```
VERSION=1.0
INFILE="c:\data\contacts.txt"
COLUMN=20,1,,,"FirstName"
COLUMN=40,21,1,,"LastName"
COLUMN=40,81,1,,"Company"
COLUMN=8,61,5,,"DateLastContacted"
OUTFILE="c:\data\cntct.wk4",WK4
```

Notice that in this fixed fielded input file, the “DateLastContacted” data actually comes before the “Company” column/field (indicated by the `startpos` parameter for each). However, since the COLUMN statement for “Company” appears third in the EDF, it will be written as the third column in the spreadsheet.

Example 3

The following EDF example translates a ASCII report input file with record-per-page data into a Paradox database:

```
ACCOUNT: 400-234-242399     PATIENT: SMITH, RONALD
INSURED: Y                  COMPANY: MEDTECH INDEMN.
POLICY#: HMDSR88900-9980    TIME ADMITTED: 11:46
SEEN BY: H. NORMAND         DIAGNOSIS CODE: HBST1
TREATMENT CODE: HBSTC3      COVERED TREATMENT: Y

ACCOUNT: 845-538-546839     PATIENT: DEAN, CHRISTINA
INSURED: Y                  COMPANY: ENSURE MEDICAL
POLICY#: CYMD800-5480       TIME ADMITTED: 13:02
SEEN BY: H. NORMAND         DIAGNOSIS CODE: RCVN04
TREATMENT CODE: RCTRZ2      COVERED TREATMENT: Y
```

To extract the Account number, patient name, policy number and diagnosis code, the EDF definition will use Reference Points and Line Tags to extract the fields.

```
VERSION=1.0
INFILE="c:\data\p-report.txt"
REFPT="ACCOUNT:",1
TAG=0,10,18,1,"PatientID"
```

```
TAG=0,38,20,1,"PatientName"
TAG=2,10,18,1,"Policy#"
TAG=3,45,6,1,"Condition"
INCLUDE="TREATMENT CODE:",1,1
OUTFILE="c:\data\patients.db",DB35
```

This EDF is specifically designed for the record-per-page type ASCII report above. Notice that no columns are defined but a `REFPT` and `TAG`s have been defined to extract data fields from the report. The `INCLUDE` statement allows the collected TAG information to be written to the output file at the end of each patient record. Without the INCLUDE statement, no data would be output.

EDF Statements

Control parameters in an EDF file have one of two functions: 1) specify the name, structure and format of an output file or 2) specify the name of a data source file and how to read data from that file. The following section details EDF controls and their general use.

VERSION (Required)

Syntax: `VERSION=version#`

Description: This required statement specifies the version of the EDF format you are using and *must be on the first line of the EDF*. This parameter is used to determine how to read the EDF statements.

The `version#` parameter for the format described in this manual is 1.0. Be sure to check the README.TXT file for any changes to this format and corresponding new version numbers.

Example:

```
VERSION=1.0
```

Related Functions: None

COLUMN (Required)

Syntax: `COLUMN=width[,startpos[,type[,dup[,"name"]]]]`

Description: This required statement defines the columns/fields to be output from the source file. One column statement must be provided for each column to be output. When using a CSV source file, the order of the column statements corresponds to the order of the fields in the source file (i.e., the first column statement defines the first column/field in the CSV, the second statement defines the second column/field, etc.).

When using a fixed fielded input file, the order in which column statements are provided is the order they are written into the output file. Therefore, the order of the columns/fields can be changed during translation. For example, if the first column statement identifies a field starting at character position 50, that column/field is output first, even though it may be the second or third column in the source file.

Note: The maximum total number of COLUMN and TAG statements cannot exceed 255.

The `width` parameter is required for any column statement. When using a CSV source file, this parameter specifies the width—in number of characters—of the column/field in the output file. When using a fixed fielded source file, this parameter specifies the width of the column/field in both the source file *and* the output file.

The `startpos` parameter is a required numeric code when translating a fixed field source file. This number identifies the position of the first character—in number of characters—(base 1) of a field in the input file.

The `type` parameter is an optional numeric code which identifies the type of data contained in the column/field. The type is a numeric code corresponding to the "StringTypes" defined in Appendix A. If this parameter is not provided, DataExport/DLL will read the first cell of the column/field in the source file, attempt to format the column as numeric values (DXString_Default = 0) and, if that fails, format it as text strings (DXString_Text = 1).

The `dup` parameter is an optional numeric code which allows you to fill-down data from previous rows/records. A value of zero or no value leaves this option off. Any other positive or negative integer activates the option (e.g., -2, -1, 1, 2). The `dup` option is used with source files where some fields are understood. For example, a source file might contain the name "FRED", in the first field of the first record and in the proceeding records, leave this field blank until records for a new person "JILL" begin. To deal with this situation, you could activate the fill-down option (value=1) to have DataExport/DLL fill "FRED" into the fields of the records below the first record which corresponds to him, then fill "JILL" into the records below the first record which corresponds to her, etc.

The `name` parameter is a string providing the name of the column/field and is only required for database and mail merge output formats. However, we recommend that you provide a name for each column/field regardless of the output format for the sake of consistency and to avoid errors when the parameter is required. DataExport/DLL simply discards a `name` if none is required and does not return an error.

Examples:

Definition for a 20-character wide column created from a CSV input file:

```
COLUMN=20,,,,"AnyField2"
```

Definition for a 40-character wide column, starting at character position 25 in a fixed fielded source file, with a text StringType, filled down and a column/field name of "LastName":

```
COLUMN=40,25,9,1,"LastName"
```

Definition for a 40-character wide column with Year/Month/Day date StringType and a column/field name of "DateField3" created from a CSV source:

```
COLUMN=10,,5,,"DateField3"
```

Related Functions: REFPT, TAG

INFILE (Required in EDF or command line)

Syntax: `INFILE="string"[,fielddel#,stringdel#]`

Description: This statement is required if no input file is specified in the `CmdLine` parameter of the `DXLaunchEngine` function. If a source file is specified in both an INFILE statement and the `CmdLine` parameter, the `CmdLine` definition takes precedence.

The `string` parameter specifies the name of the source file—either a fixed fielded or a CSV file. The `string` field specifies the name and path, if required, of the source data file to be translated.

The `fielddel#` and `stringdel#` are optional parameters used for translation of a CSV file. The `fielddel#` parameter is the ASCII number of the field delimiter (default value is 44=“,”) and the `stringdel#` is the ASCII number of the string delimiter (default value is 34=<">). The string delimiter is the character is used to enclose text strings in a CSV source file, (e.g., "1001 North St., Ste. A").

Example:

```
INFILE="c:\data\rdata1.csv",44,34
```

Related Functions: `OUTFILE`

OUTFILE (Required in EDF or command line)

Syntax: `OUTFILE="string","filetype"`

Description: This statement is required if no output file is specified in the `CmdLine` parameter of the `DXLaunchEngine` function. If an output file is specified in both an OUTFILE statement and the `CmdLine` parameter, the `CmdLine` definition takes precedence.

The `string` parameter specifies the name of the output file—either a fixed fielded or a CSV file. The `string` field also includes the full path of the output file to be written. A file extension for the file name is not required and will be provided automatically.

The `filetype` parameter is a string code that specifies the output file format using a FileType code (see “FileTypes” in Appendix A).

Examples:

```
OUTFILE="c:\data\output\my-sheet",XL5
```

```
OUTFILE="c:\data\output\my-db.mdb",MDB
```

Related Functions: INFILE

TITLE

Syntax: `TITLE=startrow,endrow`

Description: This optional statement defines lines in your source file as title rows/lines for spreadsheet or text file. This function essentially takes the lines you specify in the source file and writes them, as-is, into the first cell of the spreadsheet row that DataExport/DLL is currently writing. The `TITLE` function is useful for translating titles from ASCII reports into a recognizable equivalent in a spreadsheet. Title lines are not output to database formats.

The `startrow` parameter is a required number (base 1) that specifies the row/line where the title information starts.

The `endrow` parameter is a required number (base 1) that specifies the row/line where the title information ends.

Examples:

`TITLE=1,1`

`TITLE=2,3`

Related Functions: `HEADING`

HEADING

Syntax: `HEADING=startrow,endrow`

Description: This optional statement allows you to use lines in your source file to define headings for columns in a spreadsheet or text file. This function takes delimited or fixed field information from the source file and writes it into the top of each row in a spreadsheet as text, regardless of the definition of the data in the `COLUMN`. Heading lines are not output to database formats.

The `startrow` parameter is a required number (base 1) that specifies the row/line where the heading information starts.

The `endrow` parameter is a required number (base 1) that specifies the row/line where the heading information ends.

Examples:

`HEADING=1,1`

`HEADING=2,3`

Related Functions: `TITLE`

INCLUDE

Syntax: `INCLUDE="string",#lines[,startpos]`

Description: This optional statement is used to selectively translate lines in the source file based on the occurrence of a text string. This function is most useful when translating a report where only certain lines contain needed data. Multiple include statements can be used to select lines in the source file up to a maximum of 100 statements.

Note: Using an include statement changes the way DataExport/DLL translates a file. By default, the library translates all lines in a source file. If an INCLUDE statement is placed in the EDF, only lines which match the criteria of INCLUDE statements (or are declared as a TITLE or HEADING) will be translated.

The `string` parameter is required and specifies the alphanumeric text to be used as criteria for including lines. This parameter is case sensitive and can contain wildcard characters:

^	(caret)	Any number 0 through 9
!	(exclamation)	Any character except 0 through 9
~	(tilde)	Any character except blank
_	(underscore)	Any character including blank

These wildcards are used for single characters only. For example, in order to INCLUDE a line with the string “AC-235”, the wildcard string should read “!!-^^^”.

The `#lines` parameter is required and specifies the total number of lines to be included when the `string` parameter is found. For example, a `#lines` value of 3, will cause DataExport/DLL to translate a line which contains the `string` value, plus two additional lines after it.

The `startpos` parameter is an optional character position number that allows you to include a line only when the `string` text occurs at a specific character position. If `startpos` is zero or not defined, the INCLUDE statement applies to lines which contain the `string` anywhere on the line.

Examples:

```
INCLUDE="AREA-01”,1
```

```
INCLUDE="CHICAGO”,3,14
```

Related Functions: `COLUMN`, `TAG`, `EXCLUDE`

EXCLUDE

Syntax: `EXCLUDE="string",#lines[,startpos]`

Description: This optional statement is used to selectively ignore lines in the source file based on the occurrence of a text string. This function is most useful when translating a text report where certain lines of data are not needed. Multiple exclude statements can be used to ignore lines in the source file up to a maximum of 100 statements.

The `string` parameter is required and specifies the alphanumeric text to be used as criteria for excluding lines. This parameter is case sensitive and can contain wildcard characters:

^	(caret)	Any number 0 through 9
!	(exclamation)	Any character except 0 through 9
~	(tilde)	Any character except blank
_	(underscore)	Any character including blank

These wildcards are used for single characters only. For example, in order to EXCLUDE a line with the string “AC-235”, the wildcard string should read “!!-^^^”.

The `#lines` parameter is required and specifies the total number of lines to be excluded when the `string` parameter is found. For example, a `#lines` value of 3, will cause DataExport/DLL to ignore a line which contains the `string` value, plus two additional lines after it.

The `startpos` parameter is an optional character position number that allows you to exclude a line only when the `string` text occurs at a specific character position. If `startpos` is zero or not defined, the INCLUDE statement applies to lines which contain the `string` anywhere on the line.

Examples:

```
EXCLUDE="AREA-01",1
```

```
EXCLUDE="CHICAGO",3,14
```

Related Functions: `COLUMN`, `INCLUDE`

PAUSE

Syntax: `PAUSE="string"[,startpos]`

Description: This optional statement stops translation of a source file based on the occurrence of a text string. This function and the RESUME statement are typically used to translate text reports where only a particular section (or sections) are needed.

The `string` parameter is required and specifies the alphanumeric text to be used as criteria for stopping translation of the input file. This parameter is case sensitive and can contain wildcard characters:

^	(caret)	Any number 0 through 9
!	(exclamation)	Any character except 0 through 9
~	(tilde)	Any character except blank
_	(underscore)	Any character including blank

These wildcards are used for single characters only. For example, to create a wildcard string that will find the text string "AC-235", the wildcard string should read "!!-^^^".

The `startpos` parameter is an optional character position number that allows you to pause translation only when the `string` text occurs at a specific character position. If `startpos` is zero or not defined, the INCLUDE statement applies to lines which contain the `string` anywhere on the line.

Examples:

```
PAUSE="End of Section 2"
```

```
PAUSE="NY",5
```

Related Functions: `RESUME`

RESUME

Syntax: `RESUME="string"[,startpos]`

Description: This optional statement re-starts translation of a source file after a `PAUSE`, based on the occurrence of a text string. This function and the `PAUSE` statement are typically used to translate text reports where only a particular section (or sections) are needed.

The `string` parameter is required and specifies the alphanumeric text to be used as criteria for re-starting translation of the input file. This parameter is case sensitive and can contain wildcard characters:

^	(caret)	Any number 0 through 9
!	(exclamation)	Any character except 0 through 9
~	(tilde)	Any character except blank
_	(underscore)	Any character including blank

These wildcards are used for single characters only. For example, to create a wildcard string that will find the text string "AC-235", the wildcard string should read "!!-^^^".

The `startpos` parameter is an optional character position number that allows you to resume translation only when the `string` text occurs at a specific character position.

If `startpos` is zero or not defined, the INCLUDE statement applies to lines which contain the `string` anywhere on the line.

Examples:

```
RESUME="Section 2"
```

```
RESUME="NC",5
```

Related Functions: `PAUSE`

REFPT

Syntax: `REFPT="string",startpos`

Description: This optional statement provides a function for extracting positionally related data fields and is used exclusively for translating ASCII report files. REFPT defines a Reference Point in a report file which is used for locating and extracting data from fields. This function is used for extracting data from a report where information for a record is spread out over several lines. This type of report is equivalent to printing out records in a form or from a viewer in a database.

The REFPT function sets a text landmark for each record by identifying a text string that appears in the same visual location in the file for each record, usually in the first line of a multi-line record. After a REFPT, or Reference Point, is established, fields within a multi-line record can be defined using TAG statements.

TAG-ed fields are *not* automatically written to the output file. Once a Reference Point is found in the input file, its corresponding TAGs are read into memory and are only output when an included line (See INCLUDE above) is encountered. When this occurs, TAG-ed fields are written into their own virtual columns/fields at the beginning of the each row/record. An included line does not have to contain any defined COLUMNs in order for the TAG information to be output.

Note: A maximum of 5 REFPT statements can be made in an EDF file. TAG statements must immediately follow the REFPT statements with which they are associated. There is no limit to the number of TAGs that can be associated with a single REFPT. However, the maximum total number of TAG and COLUMN statements cannot exceed 255.

The `string` parameter is required and specifies the alphanumeric text to be used as criteria for identifying a Reference Point in the input file. This parameter is case sensitive and can contain wildcard characters:

^	(caret)	Any number 0 through 9
!	(exclamation)	Any character except 0 through 9
~	(tilde)	Any character except blank
_	(underscore)	Any character including blank

These wildcards are used for single characters only. For example, to create a wildcard string that will find the text string “AC-235”, the wildcard string should read “!!-^^^”.

The `startpos` parameter is a required character position number that specifies the first character where the `string` text occurs.

Examples:

```
REFPT="EMPLOYEE:",1
```

```
REFPT="JOB#:",5
```

Related Functions: `TAG, INCLUDE`

TAG

Syntax: `TAG=linesdown,startpos,width[,type[,"name"]]`

Description: This statement is used in conjunction with the REFPT statement for extracting positionally related data fields and is used exclusively for translating ASCII report files. When used with a REFPT statement, a TAG defines the width and character position—defined in relationship to the preceding REFPT—of a field in a set of multi-line records.

TAGs are defined in terms of the number of lines down from the preceding REFPT and character position on that line. TAGs can overlap each other and a TAG can even be defined over the REFPT it is built from.

Note: The total number of TAGs and COLUMNs cannot exceed 255.

The `linesdown` parameter is a required number that defines the number of lines down from the preceding REFPT where the TAG field begins. If the field occurs on the same line as REFPT this variable is set to zero.

The `startpos` parameter is a required character position number that specifies the first character position of the `TAG` field.

The `width` parameter is a required number which defines the total character width of the TAG field. This width should be set to the maximum width that can occur in this field.

The `type` parameter is an optional numeric code which identifies the type of data contained in the TAG. The type is a numeric code corresponding to the "StringTypes" defined in Appendix A. If this parameter is not provided, DataExport/DLL will read the first cell of the column/field in the source file, attempt to format the column as numeric values (DXString_Default = 0) and, if that fails, format it as text strings (DXString_Text = 1).

The `name` parameter is a string providing the name of the column/field for the TAG and is only required for database and mail merge output formats. However, we recommend that you provide a name regardless of the output format for the sake of consistency and to avoid errors when the parameter is required. DataExport/DLL simply discards a `name` if none is required and does not return an error.

Examples:

```
TAG=2,10,14,,"AnyField"
```

```
TAG=0,17,8,5,"DateField3"
```

Related Functions: `REFPT`, `COLUMN`

UDD

Syntax: `UDD=fielddel#[,stringdel#]`

Description: This optional statement defines the delimiters for a custom-delimited ASCII output file. This statement is only needed if you are using the UDD DataType (see "DataTypes," Appendix A) for your output file.

The `fielddel#` parameter is a required numeric code identifying the ASCII character which separates the cells/fields of data in the UDD output file. In a CSV file, this character is a comma “,”. The `fielddel#` is provided as its ASCII number without leading zeros. For example, a comma’s ASCII number is 0044 and is provided as “44”. The default value for `fielddel#` is 44.

The `stringdel#` parameter is a optional numeric code identifying the ASCII character which marks the boundaries of a text string in the UDD output file. In a CSV file, this character is a straight quotation mark <">. The `fielddel#` is provided as its ASCII number without leading zeros. For example, a straight quotation mark ’s ASCII number is 0034 and is provided as “34”. The default value for `stringdel#` is 34. If you do not require a string delimiter, either omit a value or use zero “0”.

Examples:

```
UDD=44,34
```

```
UDD=59,39
```

Related Functions: None

TABLENAME

Syntax: `TABLENAME="string"`

Description: This optional statement specifies the name of a Table in Access. This statement is only used when creating an Access output file.

The `string` parameter is a required string which contains the name of the table to be written in the Access database. Spaces are allowed and the maximum length is 64 characters. The default name for a table is “Table1” if none is specified.

Examples:

```
TABLENAME="Table_1-X"
```

```
TABLENAME="Joe’s Output Table"
```

Related Functions: None

SHEETNAME

Syntax: `SHEETNAME="string"`

Description: This optional statement specifies the name of a Sheet in an Excel workbook. This statement is only used if you are creating an Excel 5.0 or later output file.

The `string` parameter is a required string which contains the name of the sheet to be written in the Excel workbook. Spaces are allowed. The default name for a sheet is “Sheet1”.

Examples:

```
SHEETNAME="Sheet_1-X"
```

```
SHEETNAME="Rob’s Test Sheet"
```

Related Functions: None

CURRENCY

Syntax: `CURRENCY="string"`

Description: This optional statement specifies the character(s) which should be interpreted as a currency symbol. This option is useful for translation of reports with non-US currency. The default currency symbol is “$”.

The `string` parameter is a required text string containing the ASCII characters to be interpreted as a currency symbol in the input file.

Examples:

```
CURRENCY="¥"
```

```
CURRENCY="£"
```

```
CURRENCY="DM"
```

Related Functions: `THOUSAND, DECIMAL`

THOUSAND

Syntax: `THOUSAND="string"`

Description: This optional statement specifies the character used as the thousands separator in numbers over 999 (e.g., the “,” in 1,000). The default thousands separator is “,”. In many European countries, the thousands separator is a period “.” or a space “ ”.

The `string` parameter is a required text string containing the ASCII character to be interpreted as a thousands separator in the input file.

Examples:

```
THOUSAND=","
```

```
THOUSAND="."
```

Related Functions: `DECIMAL, CURRENCY`

DECIMAL

Syntax: `DECIMAL="string"`

Description: This optional statement specifies the character used as the decimal character in numbers (e.g., the “.” in 0.001). The default decimal character is a period “.”. In many European countries, the decimal separator is a comma “,”.

The `string` parameter is a required text string containing the ASCII character to be interpreted as a decimal character in the input file.

Examples:

```
DECIMAL="."
```

```
DECIMAL=","
```

Related Functions: `THOUSAND, CURRENCY`

CODEPAGE

Syntax: `CODEPAGE=codepage`

Description: This optional statement specifies the ASCII code page to be used in the interpretation of the input file. This setting is only relevant when creating Lotus 1-2-3 spreadsheets.

A code page determines how the characters above 128 are interpreted and is usually specific to a language or country. Except for Lotus spreadsheets, all file formats supported by DataExport/DLL use the system settings of the end user's machine to deal with code pages.

The `codepage` parameter is a required numeric code which identifies the code page to be used in the interpretation of a file. The default page is the US code page (437).

US ASCII	437
Multilingual	850
Portuguese	860
Canadian French	863
Nordic	865

Examples:

```
CODEPAGE=437
```

```
CODEPAGE=850
```

Related Functions: None

CUSTOMDATE

Syntax: `CUSTOMDATE="string"`

Description: This optional statement specifies how to interpret a non-delimited numeric date in the input file. Custom date interpretation is only used on column/fields in an input file that are defined as StringType 10 (DXString_Date_Custom) in the `type` parameter of a COLUMN statement.

The `string` parameter is a required string which specifies the format of a non-delimited date with a combination of the letters "Y" for year, "M" for month and "D" for day. The default custom date is YYMMDD.

Examples:

```
CUSTOMDATE="YYMMDD"
```

```
CUSTOMDATE="MMDDYYYY"
```

Related Functions: `COLUMN`

CENTURY

Syntax: `CENTURY=yearno`

Description: This optional statement defines the assumed century in a two-digit year. Any two-digit year lower that the provided value is interpreted as 20xx, while any higher value is interpreted as 19xx. For example, with the default value of “50”, the year “95” is interpreted as “1995”, however, a year of “49” is interpreted as “2049”.

The `yearno` parameter is a required number that defines the cut-off year for interpreting a date as 19xx or 20xx.

Examples:

```
CENTURY=50
```

```
CENTURY=70
```

Related Functions: `CUSTOMDATE, COLUMN`

MONTHS

Syntax: `MONTHS="string1", ... ,"string12"`

Description: This optional statement defines the month names to be used in interpreting dates. This function is useful when translating reports that contain non-US month names in dates. US month names are the default values ("January","February", ... ,"December") for months.

The `string1-12` parameters are the *full* names of months to be used in date interpretation. Full names are required for interpretation of both abbreviated months and complete month names.

Example:

```
MONTHS="January","February","March","April","May",
"June","July","August","September","October",
"November","December"
```

Related Functions: `COLUMN`

ADJUSTWIDTH

Syntax: `ADJUSTWIDTH=x`

Description: This optional statement adds one character space to the width of each column in the output file so that the data visually “fits” inside the column. This option is usually only useful for fixed field input files which are closely spaced.

The `x` parameter is a required numeric code of 0 or 1: off = 0 , on = 1. The default value is off (0).

Example:

```
ADJUSTWIDTH=1
```

Related Functions: None

SKIPMODE

Syntax: `SKIPMODE=x`

Using an INCLUDE statement in your EDF automatically sets SKIPMODE to 1 (on).

Description: This optional statement allows you to invert the way DataExport/DLL translates a file. By default, the library translates all lines in a file (`SKIPMODE=0`). If `SKIPMODE` is set to “1”, then only lines that are specifically `INCLUDE`-d will be translated.

The `x` parameter is a required numeric code of 0 or 1: 0=off , 1=on. The default value is off (0).

Example:

```
SKIPMODE=1
```

Related Functions: `INCLUDE`

STARTCELL

Syntax: `STARTCELL=startrow[,startcol[,startsheet]]`

Description: This optional statement is used to specify the cell and sheet in a spreadsheet file where DataExport/DLL begins writing data. This option is only used with spreadsheet output files. The default value is the first cell of the first sheet in the output file.

The `startrow` parameter is a required number identifying the row of the first cell to be written to in the spreadsheet.

The `startcol` parameter is an optional number identifying the column of the first cell to be written to in the spreadsheet.

The `startsheet` parameter is an optional number identifying the sheet of the first cell to be written to in the spreadsheet. This parameter is only used with the Lotus 1-2-3 v3, 4, 5 and Quattro Pro for Windows output formats.

Examples:

```
STARTCELL=3
```

```
STARTCELL=2,3
```

```
STARTCELL=4,2,2
```

Related Functions: `SHEETNAME`

SIGNEDOP

Syntax: `SIGNEDOP=leading[,"string"]`

Description: This optional statement is used to define the interpretation of mainframe-style signed overpunch numbers in an input file. Signed overpunch notation uses a single character to indicate positive and negative values 0–9. This option is only used when you have defined one or more COLUMNs with a type of 10 (DXString_SOP).

The `leading` parameter is a required numeric code which indicates that signed overpunch characters are at the beginning of numbers in the source file (1) or at the end of numbers (0). The default position is that the signed numbers are at the end (0).

The `string` parameter is an optional string which defines a custom set of signed overpunch characters. The first ten characters define the positive numbers 0–9 and the next ten characters define the negative numbers 0–9. The default value is “0123456789}JKLMNOPQR”.

Examples:

```
SIGNEDOP=0,"0123456789}JKLMNOPQR"
```

```
SIGNEDOP=1,"{ABCDEFGHI}JKLMNOPQR"
```

Related Functions: `COLUMN`

EDF Statements Quick Reference

The following is a list of EDF statements, including their parameters and their default values, if any:

Required Statements

```
VERSION=1.0

COLUMN=startpos,width[,type[,dup[,"name"]]]

INFILE="string"[,fielddel#,stringdel#]

OUTFILE="string"[,"filetype"]
```

Optional Statements

```
TITLE=startrow,endrow

HEADING=startrow,endrow

INCLUDE="string",#lines[,startpos]

EXCLUDE="string",#lines[,startpos]

PAUSE="string",startpos

RESUME="string",startpos


REFPT="string",startpos

TAG=linesdown,startpos,width[,type[,"name"]]


UDD=fielddel#[,stringdel#]  (44,34)

TABLENAME="string"          (Table1)

SHEETNAME="string"          (Sheet1)


CURRENCY="string"                  ($)

THOUSAND="string"                  (,)

DECIMAL="string"                   (.)

CODEPAGE=codepage                  (437)

CUSTOMDATE="string"         (YYMMDD)

CENTURY=yearno                     (50)

MONTHS="string1", ... ,"string12"  ("January","February", ...
,"December")


ADJUSTWIDTH=x               (0)

SKIPMODE=x                         (0)

STARTCELL=startrow[,startcol[,startsheet]]      (0,0,0)

SIGNEDOP=leading[,"string"] (0,
       "0123456789}JKLMNOPQR")
```

Appendix C: Distribution of the DataExport/DLL Libraries

This appendix describes what DataExport/DLL files are necessary for distribution with your product. Some format-specific libraries are quite large and can be omitted if disk space is a consideration. Required libraries are detailed in the “Required Distribution Files” section. Format-specific libraries are listed in the “Format Support Libraries” section.

DataExport/DLL Distributable Files

The files listed below are distributable with your product(s). Check the README.TXT file for any recent changes to this list. Distributing any of any other files provided with DataExport/DLL not listed here or indicated as distributable in the README.TXT file that came with this product is a violation of the DataExport/DLL license agreement.

Files on Diskette	Directory Installed to	Name when Installed	Description
dxtrans.dl_	Windows System	dxtrans.dll	DataExport/DLL
dxintl.dl_	"	dxintl.dll	required by DXTRANS
vb2olecf.dl_	"	vb2olecf.dll	required by DXTRANS
vbdb300.dl_	"	vbdb300.dll	required by DXTRANS
vbrun300.dl_	"	vbrun300.dll	required by DXTRANS
compobj.dl_	"	compobj.dll	OLE Compound Object DLL
storage.dl_	"	storage.dll	OLE Storage DLL
msajt112.dl_	"	msajt112.dll	Access Jet Engine

msajt200.dl_	"	msajt200.dll	Access Jet Engine
pxengwin.dl_	"	pxengwin.dll	Paradox Engine

As with all Windows shareable DLLs, these libraries should be installed into the Windows System directory (usually \windows\system). During installation you should check the internal version numbers of any existing DLLs against the ones you are installing. To implement version checking, you can use the API in VER.DLL to copy the files, or use a third party installation program that supports version checking.

The following sections explain which libraries are required for distribution and which libraries can be optionally provided for support of additional formats.

Required Distribution Files

A minimum of five DLLs must be distributed with your product in order for DataExport/DLL to be functional. These files are:

dxtrans.dll	DataExport/DLL
dxintl.dll	Support library
vb2olecf.dll	"
vbdb300.dll	"
vbrun300.dll	"

Distributing only these files with your product will allow you to output all but a few of the total number of formats supported by DataExport/DLL. Formats *not* supported by this set of libraries are Access, Excel 5.0 and Paradox. These output formats require additional libraries as explained in the next section.

Format Support Libraries

Additional libraries are required beyond the five core DLLs listed in the previous section for output of Access 1.1, 2.0; Excel 5.0 and Paradox 3.5, 4.0. The required libraries for these formats are listed below:

Access	msajt112.dll	Access Jet Engine
	msajt200.dll	Access Jet Engine
Excel 5.0	compobj.dll	OLE Compound Object dll
	storage.dll	OLE Storage dll
Paradox	pxengwin.dll	Paradox Engine

Please note that these libraries—especially those for Access—are quite large (around 625KB compressed). You may have to weigh your users' need for these formats against the cost of providing one (or two) additional distribution diskettes with your product.

Index